

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

Ansible 权威指南

李松涛 魏巍 甘捷 著

Ansible
The Definitive Guide

- 顶级运维专家联袂推荐，资深 Ansible 布道者联合撰写，辅以原理，注重实践
- 涵盖 Ansible 基础、高级技法与定制化扩展、7 个企业实战案例，以及 Web 自动化开发，解决“入门简单、深入难”问题



机械工业出版社
China Machine Press

内容简介

多名顶级运维技术专家联袂推荐，海量运维实践者、Ansible布道者联合撰写，知识全面、实践性强。

本书共三篇，14章内容。

第一篇为基础入门篇（第1章~5章），该篇着重介绍Ansible发展史、工作原理、基础元素组成，Playbook入门等，是掌握Ansible高级技巧的基石。

第二篇为高级进阶篇（第6~11章），该篇是本书重点和最大构成部分，着重结合企业实际需求场景，以大量的实际案例介绍Ansible的高级语法和实际应用技巧，涉及的技术点有Roles、Inventory、Jinja2、Galaxy等；结合的行业主流技术包括（但不限于）Zabbix、Except、MemCache、Inotify、Logio、GitLab、Docker、LNMP、Redis、MySQL、Node.js等，并提供丰富的实战案例供大家参考学习。

第三篇为Web自动化开发篇（第12~14章），该篇主要介绍如何开发Web全自动化发布界面，使用当前最流行成熟的Python语言，并结合Django前后端技术，通过Ansible celery管理后台任务队列。此篇内容从零基础部分开始介绍，逐步引导上手。

一言概之，本书涵盖Ansible基础、高级技巧、定制化扩展、多个实用企业应用案例，以及Web自动化开发，是系统学习Ansible和自动化运维的不二之选。

Ansible 权威指南

Ansible
The Definitive Guide

李松涛 魏巍 甘捷 著



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Ansible 权威指南 / 李松涛, 魏巍, 甘捷著. —北京: 机械工业出版社, 2016.11
(Linux/Unix 技术丛书)

ISBN 978-7-111-55329-8

I. A… II. ①李… ②魏… ③甘… III. 程序开发工具—指南 IV. TP311.561-62

中国版本图书馆 CIP 数据核字 (2016) 第 258615 号

Ansible 权威指南

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 高婧雅

责任校对: 殷虹

印刷: 北京文昌阁彩色印刷有限责任公司

版次: 2016 年 11 月第 1 版第 1 次印刷

开本: 186mm×240mm 1/16

印张: 21.75

书号: ISBN 978-7-111-55329-8

定价: 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

Praise 本书赞誉

以下推荐人按姓名音序排序。

随着信息时代发展，全球运维体系不断升级，灵活多变、安全稳定、自动高效的持续保障迫在眉睫。开源运动为 IT 奠定了坚实的基础环境，使得我们可以不断吮吸着其中的养分而茁壮成长。然而，为适应快速、高效运维，自动化基础设施势必成为运维必备技能。纵观自动化工具，如 Puppet、SaltStack、Func、Chef、Ansible，基于 Linux 原生 SSH（不需要 agent），并糅合众多老牌运维工具的优秀特性，集成了批量命令执行和文件处理等诸多功能。相信不少朋友已经在使用这些工具，作者也从中直接受益，并结合实战经验汇总成本书，以帮助更多热爱开源的朋友。我们坚信，集众人智慧的结晶，专注开源事业，定能让更多人享受开源运动带来的丰硕成果。而 Ansible 也将成为专业人员必备技能，这本集合基础原理和实战案例的书籍会成为运维人员必备宝典。

——马永亮，马哥教育创始人

Ansible 可以说是配置管理领域的新锐，一经推出便受到了很多运维及客户的青睐。Ansible 的架构设计简洁，上手也非常简单，学习成本很低。在我们的客户自动化方案中，考虑到安全性、稳定性、便捷性等多方面要求，我们也把对 Ansible 的兼容作为首选。非常感谢 Stanley 和其他笔者不辞辛劳地编写此书，值得大家钦佩。相信本书能给读者带来很大的收益。

——王津银（互联网运维杂谈老王），优维科技创始人

当前，云计算正在快速落地，云使资源的利用更高效，但是云只解决了系统层面资源使用的问题，业务层面的运维自动化还必须借助运维自动化工具、结合具体的业务场景来解决。在众多的自动化工具中，使用 Python 开发的 Ansible 无疑是运维人员的最爱，因为它符

合 Python 简单高效的原则。但是 Ansible 入门容易精通难。很高兴看到李松涛和他的朋友们撰写的这本书的出版，本书使快速精通 Ansible 成为可能。相信通过阅读本书，没有接触过 Ansible 的读者可以快速入门，已经在使用 Ansible 的读者可以从中学到更多知识。

——肖力，《深度实践 KVM》作者

有一种距离叫菜鸟到高手的进阶，有一种练级捷径叫活学活用《Ansible 权威指南》。本书案例通用、好使、接地气。

菜鸟得之如获至宝，稳扎稳打中轻松晋级；高手用之简洁高效，深度实践中融会贯通。

资深脚本运维有一天会发现，越做越累，正所谓：成也脚本，累也脚本。

场景化运维，可能吗？Playbook 帮你实现操作通用或者简化，把纷繁复杂的脚本变为场景中一个个的步骤，让你可以边维护边游戏，提升运维人员的工作效率。

还在为 Serverlist 的管理发愁吗？Inventory 帮你实现服务器分层管理，架构拓扑图一目了然。

还在为生成配置文件时感叹“时间都去哪了”吗？Jinja 的高效配置生成速度，让生成 1 万个复杂配置文件由 30 分钟变为 1 分钟，并且减少了业务停机时间。

本书对 Ansible 的周边扩展介绍得比较实在，理论联系实践。作者从丰富的工作经验总结出案例，详细列举了 celery、模块扩展等具体应用，让 Ansible 更加贴合实际的应用场景。

如果你想成为场景化运维人员，如果你想提升工作效率，本书就是为你量身定制的不二选择。

——张志浩，腾讯游戏运营规划专家

随着互联网和云计算的蓬勃发展，数据中心基础设施急速增加，IT 运维逐渐成为现代企业生产经营的核心，而且要求越来越高。而要实现海量系统运维和 DevOps，兼顾稳定和效率，就离不开运维自动化软件。

回顾运维自动化的发展历程，最早的运维自动化是脚本自动化，依靠 SSH 通道批量执行脚本。但人们很快就发现，每个运维人员习惯写一堆脚本，脚本的管理维护成为问题，误操作也时有发生。为了解决这个问题，Puppet、SaltStack、Ansible 等一批优秀的开源软件应运而生，运维正式进入自动化时代。

当前中国大部分数据中心还是处于“人肉运维”的时代，自动化运维的需求非常强。但对于初学者来说，要驾驭好这些软件也不容易。很多初学者会误认为运维自动化的核心是批量执行，其实不然，运维自动化的核心是配置管理，自动化只是最终效果。

Ansible 是运维自动化软件的后起之秀，发展非常快。其特点是简单易用、无代理架构，使用 Python 这样的运维语言易于二次开发，这使得 Ansible 非常适合互联网的运维场景和初学者。

本书作者之一李松涛是行业中少有的“能文能武”运维从业者，经过了腾讯海量系统运维的锻炼，又承担了 Ansible 中国“布道者”的角色，不辞辛苦地在社区和行业分享经验，最终，花费了大量心血促成了本书的诞生。“授人以鱼，不如授人以渔”，本书不但介绍了 Ansible 的基础知识，还介绍了 Ansible 的实践经验和高阶的二次开发，对读者深入理解 Ansible、构建自动化运维体系非常有帮助。

我把运维自动化分为：人肉运维、操作自动化、资源统一配置、一体化运维、运营指挥 5 个成熟度阶段，广大运维同行可以做的事情还很多。衷心祝愿李松涛再接再厉，通过著书立说和传道授业的方式，惠及更多的运维从业者，让天下没有难运维的数据中心。

——智锦，资深运维从业者，杭州云霖科技有限公司 CEO

前言 Preface

为什么要写这本书

首次接触 Ansible 是缘于一次杭州出差。当时接触互联网 3 年左右，正是技能的储备阶段，看到 Ansible 这样的新兴自动化工具不免充满好奇。当时腾讯的蓝鲸还没有出来，但 abs 脚本和 ijobs 自动化体系已经应用多年，并在整个 IEG 中心广泛应用。大型企业讲究分工精细化，各司其职，强大的自我研发能力。但伴随业绩和 KPI 的压力，很多人其实是没有多余精力关注外界技术领域的发展，尤其是游戏行业，行业自身属性对开发人员的技术能力要求非常高，前沿开源技术与业务特殊性需求并不能很好地融合，致使多数工具依赖于开发人员，整体运维体系以应用、发现、维护、服务方向为主，底层运维没有技术能力和资源协调能力为业务创造直接价值。高级运维和领导层更需着眼于高层面的业务拓展和整体运维体系规划，所以多数互联网前沿技术以技能储备的方式被引入，待机蓄力而发。

后来蓝鲸和 ijobs 融合后，在强大技术力的驱动下，运维的技术能力进一步淡化，对应的业务能力、需求发现、服务意识被强化，并提出更高的要求，DevOps 的岗位定义更加明确。蓝鲸平台类似于苹果公司的 App Store，是一个载体，只要有开发能力就可以编写自己的应用。只要应用的通用性足够高，所有业务都可以下载使用，而通用性则是开源技术最讲究的点。同时开源工具也是非常好的学习对象，往往经过简单的修改即可变成自己的产品，因此运维对开源技术的关注度越来越高，而笔者也正是在这样的背景下接触到 Ansible。

对比主流的自动化工具 SaltStack、Puppet 等，Ansible 给人最直观的感觉就是比较简单，而这也是笔者选择使用 Ansible 最重要的理由之一。因为笔者一直认为每个人精力有限，如腾讯早期的 Ops 技能培训希望个人同时兼备 Ops 和 Dev 的战略，但直到现在身边真正同时具备 Dev 和 Ops 能力于一身的人凤毛麟角。类似于 Puppet 和 SaltStack 这样的工具，高级使用均需涉及诸如 Class 类开发这样的技能才可运用，而初级运维和没有开发经验的运维掌握面向对象

技术去开发高级应用确实没有那么妥当。Ansible 早期的官网也是以 Stupid Simple 来形容其简单程度的，其前沿的去中心化思想和近期被 RedHat（红帽）官方收购的消息，也更坚定了笔者使用 Ansible 的想法。

但当时 Ansible 在国内公司应用的并不多，且其官网屡屡被破解，使得虽然自动化的理念早已家喻户晓，但国内 Ansible 的文档和社区却始终不温不火。无独有偶，笔者发现腾讯也开始在自家蓝鲸平台使用 Ansible，并结合业务进行了深入应用，所以就产生了编写一本 Ansible 书籍的想法。因此，也有了后来的 Ansible 官网中文翻译团队和本书写作团队，再后来也就有了 Ansible 中文权威网站、运维部落微信公众号、Ansible 部落微信群、Ansible 中文权威 QQ 群。更为幸运和开心的是，在坚持的过程中也遇到了一批自动化工具爱好者。<http://www.ansible.com.cn/> 将 Ansible 官网中大家日常常用的部分功能翻译成中文，所以起名为 Ansible 中文权威指南。而后 Google、Baidu 的关键字搜索结果仅次于官网，这使得我们的信心大增。这里要特别感谢马哥 Linux 团队成员的薛定谔的章鱼、guli、以马内利、黄博文、coocla、云中鹤、stanley，这些朋友们历经数月，辛勤翻译多达 5 万字文档。

在一次和朋友聊天中，朋友问到你们 Ansible 已经应用这么久，同时也有自己独立开发的界面，现在国内 Ansible 的势头虽高，但文档和书籍欠缺，何不把你们的经验总结出来分享给更多朋友呢。我当时一怔，但也有担心：一方面精力不支，另一方面老婆怀孕，我担心生活工作不能兼顾。后来在老婆的鼓励下，经肖力和黄博文兄的引荐认识了华章公司的高编辑，正式开始书籍的编写之旅。在这个过程中，很高兴又有新的伙伴骑行牛人魏巍和 Python 能力出众的甘捷陆续加入，也使得个人的压力和精力有更多的释放，书籍的内容也有更完整、丰富的互补。在整个写书过程中我们也在成立的运维部落、Ansible 公众号和 QQ 群，定期分享书籍内容，收集用户反馈和体验。到目前为止，QQ 群近 1300 人，公众号也有 2000 多人在关注。群中也专门请行业应用经验丰富的专员来解答 Ansible 的技术类问题，同时成立专门的 QA 站点，收集用户 QQ 群问题处理方案，并对积极回答问题勇于分享的朋友定期寄送礼品以示鼓励。团队很高兴也很幸运能通过这样的方式为国内 Ansible 的发展贡献自己的力量。

本书特色

从技术层面讲，运维自动化理论及思想在国内日趋成熟，自动化工具更是遍地开花。现在在运维不再纠结于没有工具可用，而是惆怅于选择何种工具。而 Ansible 正是在这样的大环境下产生，并且迅速脱颖而出。Ansible 去中心化思想和“简单就是一切”的原则也使其在运维圈快速流行。但正如所有事物一样，入门简单并不代表深入简单，这也正是本书的意义所在。

从适合读者阅读和掌握知识的结构安排上讲，本书分为“基础入门篇”“高级进阶篇”“Web

自动化开发篇”。本书在介绍新技术应用的同时更注重读者对技术的消化和接受程度，整个过程都秉承原理→练习→实战的思路，让读者轻松逐步深入，不会有生硬和突兀感。在介绍 Ansible 的核心技术应用 Playbook 章节更是不惜用 50 页左右的篇幅，通过企业实际案例讲解分析 Playbook 的使用技巧和心得。在 Ansible 企业应用实战相关章节，详细介绍 Ansible 与现今流行技术的结合使用，以及如何自我发展、自我完善技能。

在由浅入深介绍 Ansible 的同时，本书所有的应用案例按章节顺序全部上传至 GitHub，附带自研的 Web 自动化页面，也全部开源至 GitHub（同时本书写作团队收入的 20% 将捐赠给开源组织，捐赠金额和去向也会通过公众号和网站的方式对外公开）。

读者对象

- IT 网络运维工程师
- 业务运维工程师
- DevOps 技术人员
- 中小型企业无运维岗但需运维服务器的开发人员
- 虚拟化技术人员
- 对自动化理念感兴趣的技术人员

如何阅读本书

本书分为三篇，共 14 章，其中第 1 ~ 3、6、8 ~ 10 由李松涛编写，第 4、5、7、11 由魏巍编写，第 8、12 ~ 14 章由甘捷编写。

第一篇为基础入门篇（第 1 ~ 5 章），该篇着重介绍 Ansible 发展史，工作原理，基础元素组成，Playbook 入门。该部分内容虽简单，却是掌握 Ansible 高级技巧的基石，如没有接触过相关自动化工具和 Ansible，还需认真阅读。

第二篇为高级进阶篇（第 6 ~ 11 章），该篇也是本书内容的最大构成部分，着重结合企业实际需求场景，以大量的实际案例拓展介绍 Ansible 的高级语法进阶和实际应用技巧，涉及的技术点有 Roles、Inventory、Jinja2、Galaxy 等。结合的行业主流技术包括（但不限于）Zabbix、Except、MemCache、Inotify、Logio、GitLab、Docker、LNMP、Redis、MySQL、Node.js 等，并提供丰富的实战案例供大家参考学习。

第三篇为 Web 自动化开发篇（第 12 ~ 14 章），该篇内容主要针对不想购买 Tower 产品，但又有 Web 全自动化发布界面需求的人而专门撰写。该部分内容使用当前最流行成熟的 Python，并结合 Django 前后端技术，通过 Ansible celery 管理后台任务队列。虽该部分内容从零基础部

分开始介绍，逐步引导上手，但考虑时间和精力成本，建议具备一定的 Python、Django、前端基础后进行学习。

本书前 11 章，各章没有强关联，如觉得内容已掌握可跳跃式阅读，遇到不理解的地方回头再看也问题不大。从第 12 章开始为 Web 化自动开发章节，需要循序渐进地学习，建议按顺序阅读。

勘误和支持

Ansible 的发展非常快，当我们开始着手写这本书的时候 Ansible 的版本还是 1.9.4，但没过多久 2.0 稳定版本就更新出来，但 1.9 版本分支还一直在维护，随后又陆续更新了 1.9.5 和 1.9.6 的稳定版，这对我们的写作也造成一定的困扰。当时多数公司使用的还是 1.9 版本的分支，2.0 分支也陆续收到朋友们反馈各类问题。所以本书的写作过程总体还是基于 1.9 分支的基础，1.9 和 2.0 的差别主要在于 API 接口和页面开发上，后者功能模块更加完善丰富，但对于普通使用者整体差别不大，有差别的地方书中均会提到。

由于笔者的水平有限，编写时间仓促，所有的写作过程都在深夜和周末，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。如果您有更多的宝贵意见，欢迎您关注我们的公众号 linux178，或加入我们的 QQ 群：Ansible 中文权威 -2 号群（486022616），或访问我们的问答平台 <http://www.178linux.com/qa>，我们会尽量提供最满意的解答。期待能够得到你们的真挚反馈，在技术之路上互勉共进。

我想和作者聊聊

微信公众号：

linux178

或扫以下二维码



QA 公共平台：

<http://www.178linux.com/qa>

普通用户请加群：

Ansible 中文权威 -2 号群 486022616

书籍读者请加群：

中文权威读者群 577479881

致谢

感谢翻译团队在 Ansible 官网文档翻译过程中的无私付出。

感谢魏巍、甘捷两位“笔友”在我狂轰滥炸的“淫威”下坚持写作，并持续输出高质量的内容。感谢机械工业出版社华章公司的策划编辑高婧雅，在近一年的时间中始终支持我的写作。你们的鼓励和帮助引导我们顺利完成全部书稿。

特别致谢

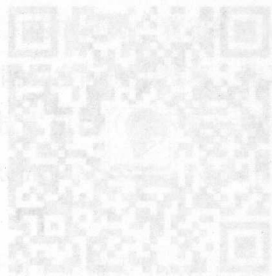
最后，我要特别感谢我的太太 yolanda，为写作这本书，我牺牲了很多陪伴她的时间，但也正因为有了她的付出与支持，我才能坚持写下去。

同时，也要郑重感谢马哥教育在我写作的过程中提供不遗余力的资源支持，让我们得以放开手脚无所束缚地完成写作工作。

谨以此书献给我最亲爱的家人，以及众多热爱开源技术的朋友们！

李松涛 (stanley)

2016 年 8 月



Contents 目 录

本书赞誉

前言

第一篇 基础入门篇

第 1 章 Ansible 基础入门 2

1.1 Ansible 是什么 2

1.2 Ansible 发展史 4

1.3 为什么选择 Ansible 5

1.4 Ansible 是如何工作的 6

1.5 Ansible 通信发展史 8

1.6 Ansible 应用场景 11

1.7 Ansible 的安装部署 12

1.7.1 PIP 方式 13

1.7.2 YUM 方式 13

1.7.3 Apt-get 方式 14

1.7.4 源码安装方式 14

1.7.5 验证安装结果 15

1.8 Python 多环境扩展管理 16

1.8.1 Pyenv 的部署与使用 16

1.8.2 Virtualenv 的部署与使用 18

1.9 本章小结 20

第 2 章 Ansible 基础元素介绍 21

2.1 Ansible 目录结构介绍 21

2.2 Ansible 配置文件解析 23

2.3 Ansible 命令用法详解 25

2.4 Ansible 系列命令用法详解与使用

场景介绍 28

2.4.1 ansible 28

2.4.2 ansible-galaxy 29

2.4.3 ansible-pull 31

2.4.4 ansible-doc 31

2.4.5 ansible-playbook 31

2.4.6 ansible-vault 32

2.4.7 ansible-console 32

2.5 Ansible Inventory 配置及详解 34

2.5.1 定义主机和组 34

2.5.2 定义主机变量 35

2.5.3 定义组变量 35

2.5.4 定义组嵌套及组变量 36

2.5.5 多重变量定义 36

2.5.6 其他 Inventory 参数列表 37

2.6 Ansible 与正则 37

2.7 本章小结 39

第3章 Ansible Ad-Hoc 命令集	40	4.4.1 限定执行范围	71
3.1 Ad-Hoc 使用场景	40	4.4.2 用户与权限设置	72
3.2 Ad-Hoc 命令集介绍	41	4.4.3 Ansible-playbook: 其他选项 技巧	73
3.2.1 Ad-Hoc 命令集用法简介	41	4.5 实战一: Ansible 部署 Node.js 企业 实践	73
3.2.2 通过 Ad-Hoc 查看系统设置	46	4.5.1 添加第三方源	73
3.2.3 通过 Ad-Hoc 研究 Ansible 的 并发特性	47	4.5.2 运行 Node.js 进程	77
3.2.4 通过 Ad-Hoc 研究 Ansible 的 模块使用	49	4.5.3 Node.js app 服务部署总结	78
3.3 Ad-Hoc 组管理和特定主机 变更	52	4.6 实战二: Drupal 基于 LAMP 的 自动化部署	78
3.3.1 Ad-Hoc 组定义	52	4.6.1 定义变量并设置 Handlers	79
3.3.2 Ad-Hoc 配置管理: 配置 Proxy 与 Web Servers 实践	54	4.6.2 部署 LAMP 基础服务	80
3.3.3 Ad-Hoc 配置后端: 配置 NoSQL 与 Database Servers 实践	56	4.6.3 配置 Apache	81
3.3.4 Ad-Hoc 特定主机变更	57	4.6.4 配置 PHP	82
3.4 Ad-Hoc 用户与组管理	58	4.6.5 配置 MySQL	83
3.4.1 Linux 用户管理	58	4.6.6 安装 Drush 和 Composer	84
3.4.2 Windows 用户管理	63	4.6.7 通过 Git 和 Drush 安装 Drupal	85
3.4.3 应用层用户管理	64	4.6.8 Drupal 部署过程总结	86
3.5 本章小结	65	4.7 实战三: Ansible 部署 Tomcat 企业实战	86
第4章 Playbook 快速入门	66	4.7.1 定义变量并设置 Handlers	86
4.1 Playbook 语法简介	66	4.7.2 安装 Java	87
4.1.1 多行缩进	67	4.7.3 安装 Tomcat 8	88
4.1.2 单行缩写	67	4.7.4 安装 Apache Solr	89
4.2 Playbook 案例分析	68	4.8 本章小结	91
4.3 Playbook 与 Shell 脚本差异 对比	71	第5章 Ansible Playbook 拓展	92
4.4 Ansible-playbook 实战小技巧	71	5.1 Handlers	92
		5.2 环境变量	93

5.3 变量	95
5.3.1 Playbook 变量	96
5.3.2 在 Inventory 文件中定义 变量	97
5.3.3 注册变量	98
5.3.4 使用高阶变量	98
5.3.5 主机变量和组变量	100
5.3.6 Facts (收集系统信息)	101
5.3.7 Ansible 加密模块 Vault	104
5.3.8 变量优先级	106
5.4 if/then/when——流程控制	107
5.4.1 Jinja2 正则表达、Python 内置 函数和逻辑判断	107
5.4.2 变量注册器 register	108
5.4.3 when 条件判断	109
5.4.4 changed_when、failed_when 条件判断	110
5.4.5 ignore_errors 条件判断	111
5.5 任务间流程控制	111
5.5.1 任务委托	111
5.5.2 任务暂停	112
5.6 交互式提示	112
5.7 Tags 标签	113
5.8 Block 块	115
5.9 本章小结	116

第二篇 高级进阶篇

第 6 章 Playbook 高级技巧进阶

6.1 巧用 Includes	118
6.1.1 Includes 使用场景	118

6.1.2 Includes 用法	119
6.1.3 动态 Includes	123
6.1.4 Handler Includes 使用 技巧	123
6.1.5 Playbooks Includes 使用 技巧	124
6.2 巧用 Roles	124
6.2.1 构建 Roles	125
6.2.2 使用 Roles 重构 Playbooks	125
6.2.3 Roles 技巧之 Handlers: 动态 变更	129
6.2.4 Roles 技巧之 Files: 文件 传输	131
6.2.5 Roles 技巧之 Templates: 模板 替换	133
6.2.6 更多复杂的跨平台 Roles	135
6.3 Jinja2 实现模板高度自定义	136
6.3.1 Jinja2 For 循环	136
6.3.2 Jinja2 If 条件	137
6.3.3 Jinja 多值合并	138
6.3.4 Jinja default() 设定	140
6.3.5 Ansible 结合 Jinja2 生成 Nginx 配置	141
6.3.6 Ansible 结合 Jinja2 生成 Apache 多主机配置	146
6.3.7 Jinja2 动态变量配置及架构 优化	148
6.4 Ansible Galaxy	151
6.4.1 Ansible-galaxy 命令用法	151
6.4.2 使用 Galaxy	152
6.5 本章小结	154

第7章 Inventory 文件扩展	155	9.3 ELK 日志系统基于 Ansible 的 自动化实现	189
7.1 Inventory 文件实战	155	9.3.1 ELK Server 的自动化实现	190
7.2 独立的 Inventory 文件	159	9.3.2 ELK Client 的自动化实现	192
7.3 Inventory 变量	159	9.4 实时日志系统基于 Ansible 的 自动化实现	192
7.3.1 host_vars 目录	160	9.4.1 配置概览	192
7.3.2 group_vars 目录	161	9.4.2 架构部署	193
7.4 动态 Inventory	161	9.5 Zabbix 基于 Ansible 的自动化 实现	195
7.5 本章小结	168	9.5.1 Zabbix Server 基于 Ansible 的 自动化实现	196
第8章 Ansible 插件扩展	169	9.5.2 Zabbix Agent 基于 Ansible 的 自动化实现	199
8.1 Ansible 插件使用场景	169	9.5.3 Zabbix Proxy 基于 Ansible 的 自动化实现	201
8.2 Ansible 插件类型	170	9.6 Ansible+Git+GitLab 实现自动化 发布	202
8.3 如何编写自己的插件	171	9.6.1 架构概览	203
8.4 插件案例实践	172	9.6.2 架构部署	203
8.5 本章小结	174	9.7 Docker 的 Ansible 自动化应用	206
第9章 Ansible 企业应用实战	175	9.7.1 Docker 容器入门	206
9.1 为新系统添加安全认证 SSHKey	175	9.7.2 使用 Ansible 创建和管理 容器	207
9.1.1 Ansible 密码认证	175	9.7.3 基于 Ansible 创建 Flask 的 Docker 容器	208
9.1.2 ssh-copy-id	176	9.7.4 数据存储容器配置	210
9.1.3 Kickstart	177	9.7.5 Flask 容器配置	211
9.1.4 Python Paramiko	178	9.7.6 MySQL 容器配置	213
9.1.5 Expect	179	9.7.7 启动容器	215
9.2 企业高可用架构的 Ansible 应用	180	9.8 本章小结	215
9.2.1 Playbook 目录编排	181		
9.2.2 高可用架构基于 Ansible 的 自动化实现	181		
9.2.3 使用 Includes 衔接各服务 配置	188		

第 10 章 Ansible 基于 Windows 的

管理架构 217

10.1 Ansible 管理机部署安装 218

10.2 Windows 系统预配置 219

10.3 Windows 下可用模块 224

10.4 Windows Ansible 模块使用 实战 224

10.5 本章小结 226

第 11 章 Ansible 安全优化篇 227

11.1 SSH 与远程连接简介 227

11.1.1 Telnet 228

11.1.2 RLOGIN、RSH 和 RCP 228

11.1.3 SSH 228

11.1.4 SSH 的发展和远程访问的 未来 229

11.2 通信加密 230

11.3 禁止 root 远程登录 231

11.4 操作系统简介 232

11.5 遵守权限最小化原则 233

11.5.1 用户管理 233

11.5.2 文件权限管理 233

11.6 定期维护更新 234

11.6.1 手动更新 234

11.6.2 自动定时更新 234

11.7 善用 Iptables 防火墙 236

11.8 定期磁盘巡检 238

11.9 系统登录日志审记 238

11.10 正确使用 SELinux 和 AppArmor 239

11.11 本章小结 240

第三篇 Web 自动化开发篇

第 12 章 Ansible 模块编写 242

12.1 初步认识 Ansible 模块 242

12.2 Ansible 简单模块编写 243

12.3 模块变量添加 245

12.4 模块状态返回的标识及应用 246

12.5 模块退出状态处理 249

12.6 模块其他功能补充 250

12.7 Ansible 模块 API 的调用 251

12.8 本章小结 265

第 13 章 开发自己的 Ansible

WebUI 267

13.1 搭建 Django 开发环境 267

13.1.1 为什么要使用 Web 页面做 管理 267

13.1.2 系统及软件环境 268

13.2 Django 配置文件详解 269

13.2.1 Django 的基础配置及 运行 269

13.2.2 Django 的主配置目录 介绍 270

13.2.3 Django 的 app 目录介绍 271

13.3 编写 Ansible 的 Web 接口 272

13.4 前端基础知识介绍 278

13.4.1 HTML 和 CSS 简介 278

13.4.2 JavaScript 简介 279

13.5 Ansible WebUI 界面开发 280

13.5.1 对接前端页面与 Ansible 的 Web 接口 280

13.5.2 配置 Web 页面传参	282
13.6 本章小结	285

第 14 章 Web 与 Ansible 结合的

常用实例	286
14.1 Web 方式管理 Ansible 的 Inventory	286
14.1.1 重新定制 Ansible 的 Hosts 文件规则	286
14.1.2 使用 ConfigParser 解析并生 成 Ansible Hosts 文件	287
14.1.3 使用数据库的存储数据生成 的 Ansible Hosts 文件	290
14.1.4 通过页面来生成 Hosts 文件	293

14.2 使用 celery 后台执行任务	301
-----------------------------	-----

14.2.1 为什么要使用 celery	301
14.2.2 使用 celery 的前期 准备	301
14.2.3 使用 celery 开始任务	303
14.2.4 使用 celery 取消正在进行的 任务	305

14.3 运行 YML 文件并实时读取 日志	306
---------------------------------	-----

14.4 通过页面上传文件并基于 Ansible 分发	313
--------------------------------------	-----

14.5 在页面上构建 YML 文件注册 中心	316
----------------------------------	-----

14.6 操作者注册中心界面	324
----------------------	-----

14.7 本章小结	331
-----------------	-----

第一篇 Part 1

基础入门篇

- 第1章 Ansible 基础入门
- 第2章 Ansible 基础元素介绍
- 第3章 Ansible Ad-Hoc 命令集
- 第4章 Playbook 快速入门
- 第5章 Ansible Playbook 拓展

命令 `ansible-galaxy` 与 `ansible` 安装包的提供, Ansible 发展非常快, 模块新增速度也非常快, 最新版的更新时间为 2016 年 8 月 31 日。

❶ 获取地址: <http://docs.ansible.com/ansible/develop/index.html>

1.1 Ansible 是什么

Ansible 是一个开源的自动化运维工具, 它可以帮助你快速部署应用, 管理配置, 以及进行其他各种任务。它的设计目标是简单易学, 易于使用, 并且能够与现有的系统无缝集成。

Ansible 基础入门

“未来主体是传统行业利用互联网技术，以云端用人工智能的方式处理大数据”，在腾讯“云+未来”技术峰会上，马化腾这样形容未来。15年前，电脑还只是少数人的专属，那时的网吧还很火，还没人知道“网咖”是什么。而现在人手一部智能手机，物联网更是让日常生活中的普通家电也能在互联网占据一席之地。这一切都推动着互联网如火如荼发展，IT 技术的发展更是日新月异，IT 工种的分类日益精细专业化。

从早期 All In One（所有应用部署在一台服务器上）的简单应用，到后期集群、高可用、缓存、消息队列、配置中心、主从分离、负载均衡、大数据存储等尖端技术的复杂应用，对运维的技术专业度和综合技能要求越来越高，运维的交付标准不再以周或天为单位，而是以分钟为单位。在现在是如此，在未来更是如此。运维不再如早期一样，手动一台台地登录服务器、部署应用配置环境、手动交付（诸如亚马逊、Google 等大型企业早已实现自动扩缩容，配置应用自动化，流程自动交付等功能）。该方式耗时耗力，很难避免人为因素的错误，最主要的是这些重复手工劳动无法让运维有更大的价值释放，这一切都是不合理的，需要有更好的解决方式。

相信看到这里，大家都明白，我们需要一套自动化工具来帮助运维更高质量、更有效地完成手头工作，以证明运维能创造的价值不止于此，况且生活不止眼前的苟且，还有诗和远方，不是吗？但当下 SaltStack、Puppet、Fabric、Chef 等自动化工具遍地开花，为什么还要推荐 Ansible 呢？读完本章你会有些许想法，未来已来，只是尚未流行。

1.1 Ansible 是什么

随着移动互联、物联网、互联网+、大数据、云计算等大规模应用的催生推动，以及人

们日常生活的互联网化，互联网的蓬勃发展不仅冲击影响着整个经济体，更对人们的生活理念影响深远。在体验到互联网带来的便利和舒适的同时，人们也不再满足于“可以用”，而是要“用得爽”，在政策、需求、利益、趋势等原因的刺激下，互联网的发展速度可想而知。众所周知，智能的背后意味着复杂，这一现象在互联网的发展中体现得淋漓尽致。在互联网迅猛发展的同时，运维这个工种也从默默无闻的后台逐步走向公众视野，被更多的人所知晓。早期公司业务有数十台、上百台服务器已经是非常庞大的规模，每个运维同时操作 10 ~ 20 台机器，忙碌地奔波于各电脑之间配置重启服务，数十人摩肩接踵的场面是何等壮观。只是每个人都在数十台机器上做同样的修改、配置、操作，如何保证每台机器的操作都完全一样呢？又如何保证其他所有人的操作都能准确无误、没有遗漏过失呢？更何况，随着互联网的迅猛发展，一个公司拥有几十台上百台机器早已不是稀奇事，巨型公司数以万计的机器都不在话下，再沿用老一套办法一台台地人工修改配置已然不现实，这该怎么办呢？相信此时你已然明白运维自动化具体是什么了。简单来讲，运维自动化就是将日常重复性的工作通过规则设定使其遵循预先既定规则，在指定的范围时间内自动化运行，但整个过程无需人工参与。而 Ansible 正是帮助运维人员实现自动化的工具之一。

Ansible 是近年越来越火的一款运维自动化工具，其主要功能是帮忙运维实现 IT 工作的自动化、降低人为操作失误、提高业务自动化率、提升运维工作效率，常用于软件部署自动化、配置自动化、管理自动化、系统化系统任务、持续集成、零宕机平滑升级等。它丰富的内置模块（如 `acl`、`command`、`shell`、`cron`、`yum`、`copy`、`file`、`user` 等，多达 569 个）^①和开放的 API 接口^②，同时任何遵循 GPL^③协议的企业或个人都可以随意修改和发布自己的版本。

Ansible 在其官网上定义如下：Ansible is a radically simple IT automation engine。即 Ansible 是一款极其简单的 IT 自动化工具。这里特别使用了 `radically simple` 来形容 Ansible 的简单程度，在 0.X 版本的 Ansible 官网中，更“过分”地使用 `Stupid Simple` 来形容 Ansible 是“令人发指的简单”。在 Ansible 官网的通篇文档中也不时使用 `Incredibly Simple`、`Keep It Simple`、`Power+Simplicity` 等字眼，可见 Ansible 这款自动化工具的设计非常注重 Simple 的理念。但 Ansible 的功能却非常不简单，完全没有因为使用方式上的简单而缩水，其自身内置模块的数量达 500 多个，而且还在快速地增加新模块，以下是这些模块的覆盖面的大致分类。

□ 系统层：支持的系统有 Linux、Windows、AIX 等，对应的模块有 `acl`、`cron`、`pip`、`easy_install`、`yum`、`authorized_key` 等大量的内置模块；

① 命令 `ansible-doc -l` 可列出所有支持的模块，Ansible 发展非常快，模块新增速度也非常快，该数据的更新时间为 2016 年 8 月 21 日。

② API 接口：<http://docs.ansible.com/ansible/developing.html>。

③ GPL/GNU GPN：GNU 通用公共协议，维基百科地址如下。<https://zh.wikipedia.org/wiki/GNU%E9%80%9A%E7%94%A8%E5%85%AC%E5%85%B1%E8%AE%B8%E5%8F%AF%E8%AF%81>

- 知名第三方平台支持：支持的云平台有 AWS、Azure、Cloudflare、Openstack、Google、Linode、Digital Ocean 等，对应的模块有 ec2、azure_rm_deployment、cloudflare_dns、clc_aa_policy、glance_image、gc_storage、digital_ocean 等；
- 虚拟化：VMware、Docker、Cloudstack、LXC、Openstack 等，对应的模块有 vmware_vmkernel、docker、cs_account、lxc_container、glance_image 等；
- 商业化硬件：F5、ASA、Citrix、Eos 等，对应的模块有 bigip_facts、asa_acl、netscaler_eos_command 等；
- 系统应用层：Apache、Zabbix、Rabbitmq、SVN、GIT 等，对应的模块有 apache2_module、zabbix_group、rabbitmq_binding、subversion、git 等。

GitHub 上有众多开源爱好者为 Ansible 贡献功能模块，这些模块完全可以满足日常工作所需。官方对模块也从使用者角度进行详细分类，如 Cloud Modules（云主机模块）、Clustering Modules（集群模块）、Commands Modules（命令模块）、Database Modules（数据库模块）等。详细的模块分类可参考官方模块列表：http://docs.ansible.com/ansible/modules_by_category.html，该网址内容的更新速度非常快，如果公司在使用 Ansible，建议最少两周关注一次。

Ansible 名字其实是来源于其作者喜欢的一本书——奥森·斯科特·卡特的《安德的游戏》，该书中 Ansible 是一种能跨越时空的即时通信工具，使用 Ansible 可以在相距数千年的距离远程实时控制前线的舰队战斗。Michael DeHaan 希望借这个名词比喻控制远端大量的服务器，因此便将自己的这款产品命名为 Ansible。

Web 界面是一款功能完善的管理工具的必备功能，Tower 是 Ansible 的 Web 化管理界面，但免费版的容量只有 10 台主机，付费版则无容量限制。基于此原因，本书的第 12 ~ 14 章会介绍搭建属于自己的 Web 化管理界面的方法，并代码全开源，具体地址大家可从 GitHub 上下载^①，同时读者遇到的问题都可以通过 QQ 群“Ansible 中文权威”咨询^②，或在微信公众号^③“运维部落”留言。

更多信息请参考：

- Ansible 官方地址：<https://docs.ansible.com/>；
- GitHub 地址：<https://github.com/ansible/ansible/blob/devel/docsite/rst/index.rst>；
- Ansible 中文权威地址：<http://www.ansible.com.cn/>。

1.2 Ansible 发展史

Ansible 的第一个版本是 0.0.1，发布于 2012 年 3 月 9 日，其作者兼创始人是 Michael

① <https://github.com/stanleyist/ansibleUI>

② QQ1 群：372011984（满）QQ2 群：486022616

③ 微信公众号 ID：linux178

DeHaan。Michael DeHaan 曾经供职于 Puppet Labs、RedHat、Michael，在配置管理和架构设计方面有丰富的经验。其在 RedHat 任职期间主要开发了 Cobble，经历了各种系统简化、自动化基础架构操作的失败和痛苦，在尝试了 Puppet、Chef、Cfengine、Capistrano、Fabric、Function、Plain SSH 等各式工具后，决定自己打造一款能结合众多工具优点的自动化工具，Ansible 由此诞生。

其第一个版本号被非常谨慎地定义为 0.01。到目前为止共发布 107 个版本，最新稳定版是 Stable 2.1.1.0-1，最新 Beat 版 Beat 2.1.1.0-0.1.rc1。值得一提的是，作为自动化工具新秀，Ansible 已被 RedHat 官方收购，在 GitHub 上被关注的势头也极为迅猛，Star 和 Fork 数是当下红极一时的 SaltStack 的 2 倍多。同类自动化工具 GitHub 关注程度如表 1-1 所示。从表可以看出 Ansible 的受欢迎度。被 RedHat 收购后，其未来发展潜力更是不可估量。

表 1-1 同类自动化工具 GitHub 关注程度 (2016-07-10)

同类自动化工具	Watch (关注)	Star (点赞)	Fork (复制)	Contributors (贡献者)
Ansible	1387	1 7716	5356	1428
SaltStack	530	6678	3002	1520
Puppet	463	4044	1678	425
Chef	383	4333	1806	464
Fabric	379	7334	1235	116

Ansible 版本更新速度非常快，有时会一天推出多个 Dev 版本，7 天推出一个稳定版本。所以使用 Ansible 的过程中也需多留意官网的更新。

1.3 为什么选择 Ansible

Ansible 自 2012 年发布以来，没多久便在美国开始流行。快速被 IT 人员接受的原因较大方面得益于 Michael DeHaan 在美国 IT 圈的名气和影响力。随后它逐步在各国流行。而笔者选择 Ansible 的原因主要有如下几个方面：

- ❑ Ansible 完全基于 Python 开发，而 DevOps 在国内已然是一种趋势，Python 被逐步普及，运维人员自己开发工具的门槛逐步降低，得益于此，方便对 Ansible 二次开发；
- ❑ Ansible 丰富的内置模块，甚至还有专门为商业平台开发的功能模块，近 600 个模块完全可以满足日常功能所需；
- ❑ 在 Ansible 去中心化概念下，一个简单的复制操作即可完成管理配置中心的迁移；
- ❑ Agentless (无客户端)，客户端无需任何配置，由管理端配置好后即可使用，这点非常诱人。在第 10 章会介绍如何部署配置 Windows 系统的主机端，用后会有深切的感受。

自 Ansible 发布后，也陆续被 AWS、Google CloudPlatform、Microsoft Azure、Cisco、HP、VMware、Twitter 等大公司接纳并投入使用。关于笔者个人与 Ansible 的一些故事，有兴趣的朋友可以参考本书前言。

1.4 Ansible 是如何工作的

Ansible 没有客户端，因此底层通信依赖于系统软件，Linux 系统下基于 OpenSSH 通信，Windows 系统下基于 PowerShell，管理端必须是 Linux 系统，使用者认证通过后在管理节点通过 Ansible 工具调用各应用模块将指令推送至被管理端执行，并在执行完毕后自动删除产生的临时文件。Ansible 具体的工作机制官方有专栏介绍 <https://www.ansible.com/how-ansible-works>，但整体稍过简略。我们参考从官网视频中的截图来详细了解下其工作方式，如图 1-1 所示。根据 Ansible 使用过程中的不同角色，我们将其分为：

- 使用者

- Ansible 工具集

- 作用对象

(1) 使用者

如图 1-1 中 Ansible 工作机制所示，Ansible 使用者来源于多种维度，图中为我们展示了 4 种方式：

第一种方式：CMDB (Configuration Management Database, 配置管理数据库)，CMDB 存储和管理着企业 IT 架构中的各项配置信息，是构建 ITIL 项目的核心工具，运维人员可以组合 CMDB 和 Ansible，通过 CMDB 直接下发指令调用 Ansible 工具集完成操作者所希望达成的目标；

第二种方式：PUBLIC/PRIVATE 方式，Ansible 除了丰富的内置模块外，同时提供丰富的 API 语言接口，如 PHP、Python、PERL 等多种当下流行语言，基于 PUBLIC (公有云)/PRIVATE (私有云)，Ansible 以 API 调用的方式运行；

第三种方式：USERS 直接使用 Ad-Hoc 临时命令集调用 Ansible 工具集来完成任务执行，Ad-Hoc 将在第 3 章有详细介绍；

第四种方式：USERS 预先编写好的 ANSIBLE PLAYBOOKS，通过执行 Playbooks 中预先编排好的任务集按序完成任务执行。

(2) Ansible 工具集

ansible 命令是 Ansible 的核心工具，ansible 命令并非自身完成所有的功能集，其只是 Ansible 执行任务的调用入口，大家可心理解为“总指挥”，所有命令的执行通过其“调兵遣将”最终完成。ansible 命令共有哪些兵将可供使唤呢？大家看中间绿色框中有 INVENTORY (命令执行的目标对象配置文件)、API (供第三方便程序调用的应用程序编

程接口)、MODULES (丰富的内置模块)、PLUGINS (内置和可自定义的插件) 这些可供调遣。

(3) 作用对象

Ansible 的作用对象，不仅仅是 Linux 和非 Linux 操作系统的主机 (HOSTS)，同样也可以作用于各类公有云 / 私有云，商业和非商业设备的网络设施。

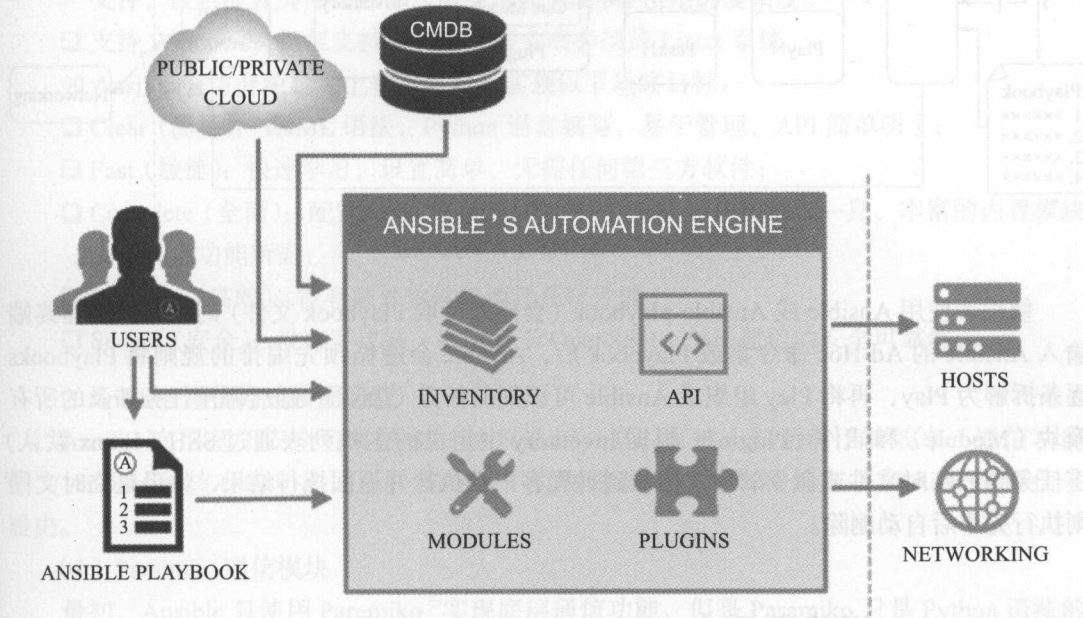


图 1-1 Ansible 工作机制

同样，如果我们按 Ansible 工具集的组成来讲，由图 1-1 可以看出 Ansible 主要由 6 部分组成。

- ❑ ANSIBLE PLAYBOOKS：任务剧本（任务集），编排定义 Ansible 任务集的配置文
件，由 Ansible 顺序依次执行，通常是 JSON 格式的 YML 文件；
 - ❑ INVENTORY：Ansible 管理主机的清单；
 - ❑ MODULES：Ansible 执行命令的功能模块，多数为内置的核心模块，也可自定义；
 - ❑ PLUGINS：模块功能的补充，如连接类型插件、循环插件、变量插件、过滤插件等，
该功能不常用。
 - ❑ API：供第三程序调用的应用程序编程接口；
 - ❑ ANSIBLE：该部分图中表示的不明显，组合 INVENTORY、API、MODULES、PLUGINS
的绿框大家可以理解是 Ansible 命令工具，其为核心执行工具；
- Ansible 执行任务，这些组件相互调用关系如图 1-2 所示：

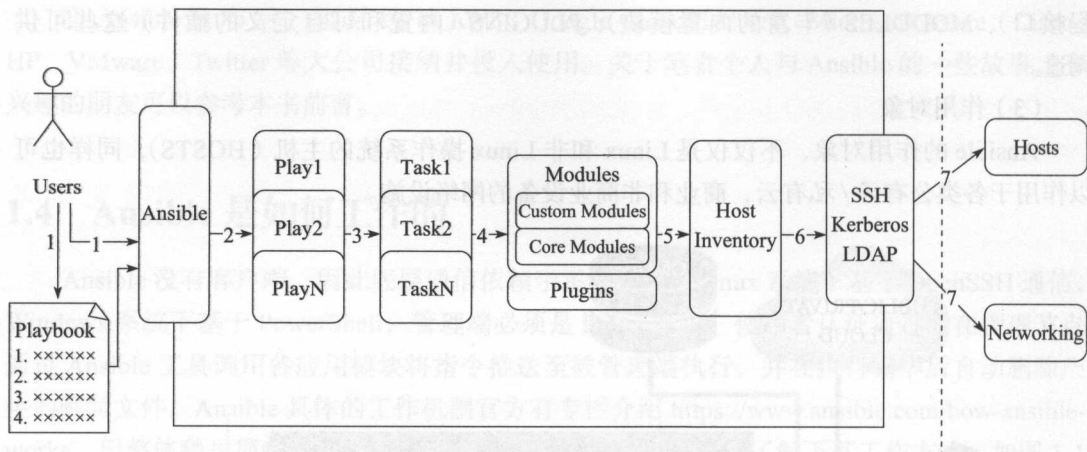


图 1-2 Ansible 组件调用关系

使用者使用 Ansible 或 Ansible-playbook（会额外读取 Playbook 文件）时，在服务器终端输入 Ansible 的 Ad-Hoc 命令集或 Playbook 后，Ansible 会遵循预先编排的规则将 Playbooks 逐条拆解为 Play，再将 Play 组织成 Ansible 可识别的任务（Task），随后调用任务涉及的所有模块（Module）和插件（Plugin），根据 Inventory 中定义的主机列表通过 SSH（Linux 默认）将任务集以临时文件或命令的形式传输到远程客户端执行并返回执行结果，如果是临时文件则执行完毕后自动删除。

1.5 Ansible 通信发展史

Ansible 主推的卖点是其无需任何 Daemon 维护进程即可实现相互间的通信，且通信方式是基于业内统一标准的安全可靠的 SSH 安全连接。同时因为 SSH 是每台 Linux 主机系统必装的软件，所以 Ansible 无需在远程主机端安装任何额外进程，即可实现 Agentless（无客户端），进而助力其实现去中心化的思想。尽管稳定、快速、安全的 SSH 连接是 Ansible 通信能力的核心，但 SSH 的连接效率一直被诟病，所以 Ansible 的通信方式和效率在过去的数年中也在不停地改变和提高。基于以上认识，我们先来了解 Ansible SSH 的工作机制，再来回顾其发展史。

1. Ansible SSH 工作机制

Ansible 执行命令时，通过其底层传输连接模块，将一个或数个文件，或者定义一个 Play 或 Command 命令传输到远程服务器 /tmp 目录的临时文件，并在远程执行这些 Play/Command 命令，然后删除这些临时文件，同时回传整体命令执行结果。这一系列操作在未来的 Ansible 版本中会越来越简单、直接，同时快速、稳定、安全。通过了解其工作机制及其一直以来秉承的去中心化思想，我们可以总结，Ansible 是非 C/S 架构，自身没有 Client 端，

其主要特点如下。

- ❑ 无客户端，只需安装 SSH、Python 即可，其中 Python 建议版本为 2.6.6 以上。
 - ❑ 基于 OpenSSH 通信，底层基于 SSH 协议（Windows 基于 PowerShell）。
 - ❑ 支持密码和 SSH 认证，因可通过系统账户密码认证或公私钥认证，所以整个过程简单、方便、安全。建议使用公私钥方式认证，因为密码认证方式的密码需明文写配置文件，虽然配置文件可加密，但会增加 Ansible 使用的复杂度。
 - ❑ 支持 Windows，但仅支持客户端，服务端必须是 Linux 系统。
- 如 Ansible 官方介绍，如上特性是希望实现以下最终目标：
- ❑ Clear（简易）：YAML 语法，Python 语言编写，易于管理，API 简单明了；
 - ❑ Fast（敏捷）：快速学习，设置简单，无需任何第三方软件；
 - ❑ Complete（全面）：配置管理、应用部署、任务编排等功能集于一身，丰富的内置模块满足日常功能所需；
 - ❑ Efficient（高效）：没有额外软件包消耗系统性能；
 - ❑ Secure（安全）：没有客户端，底层基于 OpenSSH，保证通信的安全可靠性。

2. Ansible 通信方式发展历程

Ansible 底层基于安全可靠的 SSH 协议通信，但一直被人们诟病于其效率，通信功能作为 Ansible 最核心的功能之一，官方也一直在做改进。本节我们来了解 Ansible 通信发展史。

（1）Paramiko 通信模块

最初，Ansible 只使用 Paramiko^①实现底层通信功能，但是 Paramiko 只是 Python 语法的一个第三方库，发展速度远不及 OpenSSH^②。同时，Paramiko 的性能和安全性较 OpenSSH 稍逊一筹（在笔者眼里）。

在后续发布的新版本中，Ansible 仍继续兼容 Paramiko，甚至在诸如 RHEL 5/6 等不支持 ControlPersist^③（只在 OpenSSH 5.6+ 版本中支持）的系统中封装其为默认通信模块。

（2）OpenSSH

从 Ansible 1.3 版本开始，Ansible 默认使用 OpenSSH 连接实现各服务器间通信，以支持 ControlPersist（持续管理）。Ansible 从 0.5 版本起即支持 OpenSSH 功能，但直到 1.3 版本开始才将其设置为默认。

多数本地 SSH 配置参数，诸如 Hosts、公私钥文件等是默认支持的，但如果希望通过非默认的 22 端口运行命令等操作，则需要在 Inventory 文件中配置 `ansible_ssh_port` 的值。OpenSSH 相比 Paramiko 更快、更可靠。

① Paramiko 是基于 Python 语言通过 SSH2 的开源软件。

② 基于标准 SSH 协议的实现的应用遍布开源界。

③ ControlPersist 功能允许 SSH 连接在 SSH Config 配置的过期时间内长期保持存活，以便使常用命令的执行不必每次都经过最初的握手过程。

(3) 加速模式

据官网介绍：开启加速模式后 Ansible 通信速度有质的提升，是开启 ControlPersist 后的 SSH 的 2 ~ 6 倍，是 Paramiko 通信速度的 10 倍。

尽管加速模式对 Ad-Hoc 命令不友好，但是 Playbook 通过加速模式会收到更高的性能。加速模式抛弃 SSH 多次连接的方式，通过 SSH 初始化后，带着 AES key 的初始化连接信息通过特定的端口（默认 5099，但可配置）执行命令传输文件。使用加速模式唯一需要的额外包是 python-keyczar，如此一来，几乎所有的常规模块 OpenSSH/Paramiko 均工作在加速模式，但如下使用 sudo 的情况例外：

1) sudoers 文件需要关闭其中的 requiretty 功能，注释掉或者设置每个用户的默认值为 username !requiretty。sudoers 的 man 文档说明如下。

```
requiretty
```

```
If set, sudo will only run when the user is logged in to a real tty. When this flag is set, sudo can only be run from a login session and not via other means such as cron(8) or cgi-bin scripts. This flag is off by default.
```

2) 开启加速模块必须事先设置 sudo 文件 NOPASSWD 配置，禁用 sudo 后的 PASSWORD 交互认证过程。加速模式相对 OpenSSH 可以提供 2 ~ 4 倍的性能提升（尤其对于文件传输功能），在 Playbook 的应用中可以通过增加配置开关来实现。

```
---
```

```
- hosts: all
```

```
accelerate: true
```

```
accelerate_port: 5099
```

```
[...]
```

其中的端口也可以在 ansible.cfg 中单独配置。

```
[accelerate]
```

```
accelerate_port = 5099
```

加速模式是现在已经被废弃的 Fireball 模式的进化版，类似于 Ansible 加速通信方式，但需控制机事先安装 ZeroMQ 服务（这与 Ansible 简单、无依赖、无 Daemon 的理念是相违背的），并且一点也不支持 sudo 操作。

(4) Faster OpenSSH in Ansible 1.5+

Ansible 1.5+ 版本中的 OpenSSH 有了非常大的改进，旧版本中实现方式是复制文件至远程服务器后运行，然后删除这些临时文件，而新版本的替代方案是通过 OpenSSH 发送执行命令，将所有操作附带在 SSH 连接过程中同步实现。该方式只在 Ansible 1.5+ 版本有效，且需在 /etc/ansible/ansible.cfg 的 [ssh_connection] 区域开启 pipelining=True 功能。

关于加速模式我们还需要关注如下内容：

□ pipelining=True 需结合 sudo 的 requiretty 配置方可生效，请确保 /etc/sudoers 的 Defaults requiretty 为注释状态。绝大多数现有流行系统默认开启该选项。

□ 在 Mac OSX、Ubuntu、Windows 的 Cygwin 或其他流行 OS 最好选择 5.6 以上的 OpenSSH 版本，这些版本对 ControlPersist 有更友好的支持。

□ 如 Ansible 运行的主机系统是 RHEL 或 CentOS，然而希望升级当前 OpenSSH 到最新版本以支持 Faster/Persistent 连接模式，可以通过 `yum update openssh` 升级最新版本。

本节内容可以通过如图 1-1 所示的 Ansible 通信方式发展史鱼骨图来概括，从最开始的 Paramiko，后来初步演变为 OpenSSH，加速模式官方推荐 Pipelining 方式。

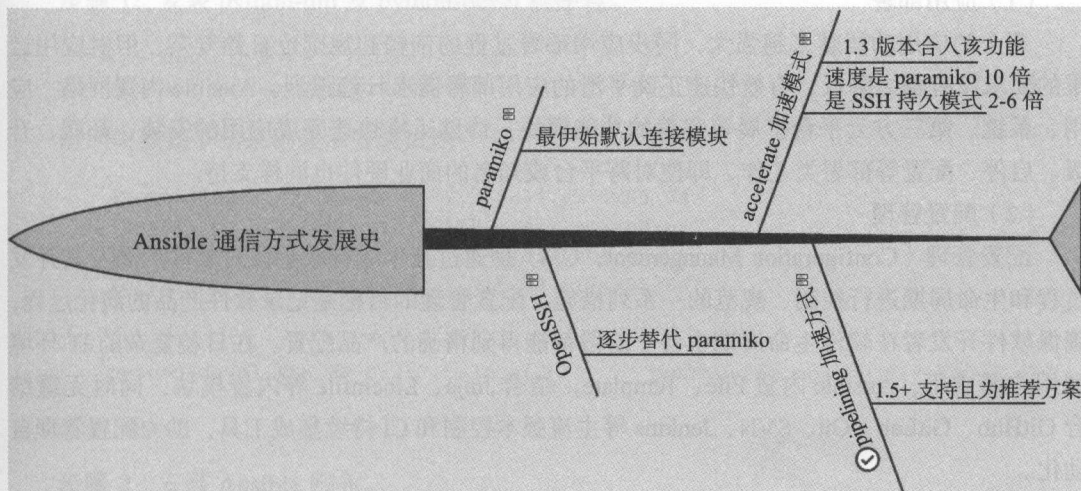


图 1-3 Ansible 通信方式发展史

在了解 Ansible 通信原理及发展史后，我们进一步学习 Ansible 自身的发展史。要知道，在 Chef、Puppet、SaltStack、Fabric 等自动化工具群雄争霸的市场背景下，Ansible 依然能够杀出重围，在 GitHub 取得如此惊人成就，并且被红帽官方收购，其发展史不得不让人刮目相看。

1.6 Ansible 应用场景

Ansible 底层基于 Python，以简单著称，配置文件格式也以 INI 和 YAML 为主，与其他管理工具相比，学习成本较低，学习曲线也很平滑，无论是基础运维人员还是资深运维工程师都可以较快上手，稍加练习便可以熟练掌握。如果具备 Dev 基础，熟悉 Python、PHP 等主流语言，基于 Ansible 开放 API 接口做二次开发，可以灵活有效地发挥其价值。Ansible 自身也包括非常丰富的内置模块，从 Windows 系统到开源 Linux 系统，从文件同步到命令执行，从软件的安全升级到配置的维护变更，从商业硬件 A10、F5 到公（私）有云 AWS、Digital、VMware、Docker 等，几乎囊括了运维日常所有的技术应用。系统下所有的操作可从运维操作角度划分为两类。

□ 文件传输：文件的本地传输和异地传输，所有文件的空间形态、时间形态变化均构成文件传输类操作。

□ 命令执行：终端所有操作对系统来讲都是指令的组成，最终转换为基础硬件可接受的电信号完成任务集。对运维操作的用户行为来讲，除文件传输以外的其他操作均可称为命令执行。

从自动化工作类型角度归类如下。

（1）应用部署

现今的应用功能越来越强大，同步应用部署过程的依赖和规则也日趋复杂，但对应用运维的要求没有随之降低，有效快速正确平滑的应用部署需求日趋强烈。Ansible 内置网络、应用、系统、第三方云平台扩展等完善的功能模块，协助运维快速完成应用的安装、卸载、升级、启停、配置等部署类工作，即使对跨平台或知名的商业硬件也同样支持。

（2）配置管理

配置管理（Configuration Management, CM）是通过技术或行政手段对软件产品及其开发过程和生命周期进行控制、规范的一系列措施。配置管理的目标是记录软件产品的演化过程，确保软件开发者在软件生命周期中各个阶段都能得到精确的产品配置。在日益复杂的 IT 环境和用户需求下，Ansible 内置 File、Template，结合 Jinja、Lineinfile 等内置模块，同时无缝结合 GitHub、GitLab、Git、SVN、Jenkins 等主流版本控制和 CI 持续集成工具，助力配置管理自动化。

（3）任务流编排

有效保证 Tasks 任务流按既定规则和顺序完成事先制订的目标和计划，同时 Roles 编排方式又能在一定程度上从书写习惯和代码层编排上保证整体项目的可架构性和规范性，协助控制项目维护成本不致过高。

如上场景适用于网络管理员、系统运维、应用运维、桌面运维、DevOps、基础架构运维等多领域运维行业，以及无运维岗但服务规模又需有一定精力投入维护的小型公司，开发人员经过简单的了解即可初步上手。同样也适用于中大型公司，可以投入人力、精力、财力对 Ansible 进行二次开发，使其更加适用。

1.7 Ansible 的安装部署

了解完 Ansible 是什么、通信原理及发展史、Ansible 发展历程及其应用场景后，接下来为大家介绍 Ansible 的安装部署。

Ansible 的安装部署非常简单，其仅依赖于 Python 和 SSH，而系统默认均已安装。除 Windows 外，RedHat、Debian、CentOS、OSX^①均可作为管理节点部署 Ansible。Ansible

① 各类类 UNIX 系统。

被 RedHat 红帽^①官方收购后，其安装源被收录在 EPEL 中，如已安装 EPEL 可直接 YUM 或 APT 安装，通过 pip 和 easy_install 的 Python 第三方包管理器也可以便捷安装 Ansible，下面我们详细介绍部署方式。

1.7.1 PIP 方式

Ansible 底层也是基于 Python 编写，所以可以通过 PIP 方式安装 Ansible。

步骤 1：安装 python-pip 及 python-devel 程序包。

```
// 安装 python-pip 程序包及 python-devel,
yum install python-pip python-devel -y
```

返回类似如下结果则表示安装成功：

```
Installing : python-devel-2.7.5-34.el7.x86_64                1/2
Installing : python-pip-7.1.0-1.el7.noarch                  2/2
Verifying  : python-pip-7.1.0-1.el7.noarch                  1/2
Verifying  : python-devel-2.7.5-34.el7.x86_64                2/2

Installed:
  python-devel.x86_64 0:2.7.5-34.el7          python-pip.noarch 0:7.1.0-1.el7

Complete!
```

步骤 2：安装 Ansible 服务。

// 安装前请确保服务器的 gcc、glibc 开发环境均已安装，系统几乎所有的软件包编译环境均基于 gcc，如不确认可先执行如下命令：

```
yum install gcc glibc-devel zlib-devel rpm-build openssl-devel -y
// 升级本地 PIP 至最新版本
pip install --upgrade pip
// 安装 Ansible 服务
pip install ansible --upgrade
```

执行命令 `ansible --version`，有类似如下返回结果则表示 Ansible 安装成功并可正常使用。

```
ansible 2.1.1.0
  config file = /etc/ansible/ansible.cfg
  configured module search path = Default w/o overrides
```

如下其他验证安装是否成功的方式也一样，均可执行 `ansible—version` 验证，后面不一一列出。

1.7.2 YUM 方式

YUM (Yellow dog Updater, Modified) 是一个在 Fedora 和 RedHat 以及 CentOS 中的

① Linux 发行商之一。

Shell 前端软件包管理器。基于 RPM 包管理，能够从指定的服务器自动下载 RPM 包并且安装，可以自动处理依赖性关系，并且一次安装所有依赖的软件包，无需烦琐地一次次下载、安装。YUM 安装 Ansible 过程如下：

```
// 需事先安装 EPEL⊖源后方可找到并安装 Ansible
rpm -Uvh https://dl.fedoraproject.org/pub/epel/epel-release-latest-6.noarch.rpm
// 安装 Ansible
yum install ansible -y
```

安装速度视网络情况而定，因为安装过程会安装非常多的依赖包，又因各系统环境的差异性，如返回类似如下结果则表示安装成功：

```
Installed:
  ansible.noarch 0:2.1.1.0-1.el7

Dependency Installed:
  PyYAML.x86_64 0:3.10-11.el7          libtomcrypt.x86_64 0:1.17-23.el7
  libtommath.x86_64 0:0.42.0-4.el7      python-babel.noarch 0:0.9.6-8.el7
  python-httplib2.noarch 0:0.7.7-3.el7   python-jinja2.noarch 0:2.7.2-2.el7
  python-keyczar.noarch 0:0.71c-2.el7    python-markupsafe.x86_64 0:0.11-10.el7
  python-pyasnl.noarch 0:0.1.6-2.el7     python2-crypto.x86_64 0:2.6.1-9.el7
  python2-ecdsa.noarch 0:0.13-4.el7      python2-paramiko.noarch 0:1.16.1-1.el7
  sshpass.x86_64 0:1.05-5.el7

Complete!
```

1.7.3 Apt-get 方式

Apt-get 全称是 Advanced Package Tool，是一款适用于 UNIX 和 Linux 系统的应用程序管理器，适用于 Ubuntu、Debian 等 deb 包管理式的操作系统，主要用于自动地从互联网的软件仓库中搜索、安装、升级、卸载软件或操作系统。

```
// 添加 Ansible 源
apt-add-repository -y ppa:ansible/ansible
// 升级库文件
apt-get update
// 安装 Ansible
apt-get install -y ansible
```

1.7.4 源码安装方式

源码安装本身就是一道很高的门槛，作为刚接触 Linux 的新手不建议使用该方式。

在什么情况下我们需要从源代码安装软件呢？其实源码安装是相对于二进制安装而言的，所谓的二进制安装即前言讲到的，PIP、YUM、Apt-get 都是二进制的安装方式，一般当

⊖ EPEL：EPEL（Extra Packages for Enterprise Linux，企业版 Linux 的额外软件包）是 Fedora 小组维护的一个软件仓库项目，为 RHEL/CentOS 提供它们默认不提供的软件包。

新软件推出了新的版本，而所用的发行版并没有及时跟进，这时候，想要“尝鲜”的话，就非得靠自己而不可使用源码编译安装；另一种情形是，不管是软件的开发者还是现用的系统都没有提供可直接使用的二进制包，而自己又非要使用该软件，那么也需源码安装才行。当然，还有其他的情形。总而言之，学会源码安装软件方式是一项非常重要的技能，但又因其编译环境准备起来复杂不堪，同时安装过程又需人工逐一解决安装过程中可能遇到的各项应用层依赖和系统库依赖，所以门槛较高，故不建议初学者使用该方式。

```
// 不建议安装 Beta 版
// 安装 Git①客户端
yum install git -y
```

整个安装过程无报错，有类似如下返回结果则表示安装成功。

```
Installed:
  git.x86_64 0:1.8.3.1-5.el7

Dependency Installed:
  libgnome-keyring.x86_64 0:3.8.0-3.el7 perl-Error.noarch 1:0.17020-2.el7
  perl-Git.noarch 0:1.8.3.1-5.el7 perl-TermReadKey.x86_64 0:2.30-20.el7

Complete!
```

安装 Ansible 软件包。

```
// 使用 Git 将拉取指定的 Ansible 版本至本地当前目录
git clone git://github.com/ansible/ansible.git --recursive
// 切换至程序包目录
cd ./ansible
// 执行 env-setup 脚本，安装 Ansible 软件包
source ./hacking/env-setup
```

1.7.5 验证安装结果

如上列举了互联网主流系统的 Ansible 安装方式，如整个过程均无报错，则执行如下命令应有类似结果返回：

```
ansible --version
ansible 1.9.6
```

如上述命令能正常执行，表示 Ansible 安装成功，并可正常使用。通常情况下，Ansible 的安装简单顺利，但确实会有安装报错的情况发生，多数情况是由本地复杂的系统环境导致的。下面我们为大家介绍 Python 多环境管理，来解决该类问题。

① Git：Git 是一款免费、开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目。Git 可以有效、高速地处理从很小到非常大的项目版本管理。Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。

1.8 Python 多环境扩展管理

众所周知，Python 发展至今，版本众多，部分版本功能差异较大，在使用过程中经常遇到第三方库依赖的 Python 版本和系统 Python 版本不一致的情况。同时又因系统底层需调用当前版本 Python，所以不能随意变更当前系统 Python 版本。如此情景下就会有 Python 多版本共存的情况。于是，Python 多环境管理工具应运而生。这里为大家介绍两款工具，分别是 Pyenv 和 Virtualenv。Pyenv 和 Virtualenv 均为 Python 管理工具，不同的是，前者是对 Python 的版本进行管理，实现不同版本间的切换和使用；而后者则通过创建虚拟环境，实现与系统环境以及其他 Python 环境的隔离，避免相互干扰。

1.8.1 Pyenv 的部署与使用

Pyenv 是一个简单的 Python 版本管理工具，以前叫作 Pythonbrew。它让你能够方便地切换全局 Python 版本，安装多个不同的 Python 版本，设置独立的某个文件夹或者工程目录特有的 Python 版本，同时创建 Python 虚拟环境（virtualenv's）。所有这些操作均可以在类 UNIX 系统的机器上（Linux 和 OS X）不需要依赖 Python 本身执行，而且它工作在用户层，不需要任何 sudo 操作。

（1）部署

Pyenv 作为 Python 的版本管理工具，通过改变 Shell 的环境变量来切换不同的 Python 版本，以达到多版本共存的目的。该工具不支持 Windows 系统。具体工作原理如下。

1) Pyenv 安装后会在系统 PATH 中插入 shims 路径，每次执行 Python 相关的可执行文件时，会优先在 shims 里寻找 Python 路径 `~/pyenv/shims:/usr/local/bin:/usr/bin:/bin`；

2) 系统选择 Python 版本，依如下顺序选择 Python 的版本：

❑ Shell 变量设置（执行 `pyenv shell` 查看）

❑ 当前可执行文件目录下的 `.python_version` 文件里的版本号（执行 `pyenv shell` 查看）

❑ 上层目录查询找到的第一个 `.pyenv-version` 文件

❑ 全局的版本号在 `~/pyenv/version` 文件内（执行 `pyenv global` 查看）

3) 确定版本文件的位置和 Python 版本后，Pyenv 会根据版本号在 `~/pyenv/versions/` 文件夹中查找对应的 Python 版本。执行命令 `pyenv versions` 可查看系统目前安装的 Python 版本。

接下来开始部署 Pyenv，具体部署方式如下：

```
// clone pyenv 至家目录
git clone git://github.com/yyuu/pyenv.git ~/.pyenv
// 修改环境变量
echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.bashrc
echo 'export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.bashrc
echo 'eval "$(pyenv init -)"' >> ~/.bashrc
```

```
// 重启当前 Shell
exec $Shell -l
```

执行 `pyenv versions` 命令，有类似如下返回结果表示安装正常：

```
* system (set by /home/o2o/.pyenv/version)
  2.7.8
```

接下来我们来了解 Pyenv 的使用方式。

(2) 通过 Pyenv 管理多 Python 版本

Pyenv 命令使用规则如下：

```
Usage: pyenv <command> [<args>]
```

我们通过 Pyenv 安装 Python 3.4.1 版本来熟悉其用法。

```
// 查看可安装的版本列表
pyenv install -list
// 安装指定的 Python 版本
pyenv install 3.4.1
// 切换当前目录 Python 版本为 3.4.1
pyenv local 3.4.1
// 切换全局目录 Python 版本为 3.4.1
pyenv global 3.4.1
// 刷新 shims
pyenv rehash
```

Pyenv 更多用法如下：

commands	列出 pyenv 的所有可用命令
local	设置或列出当前环境下 Python 版本号
global	设置或列出全局环境下 Python 版本号
shell	设置或列出 Shell 环境下 Python 版本
install	安装指定的 Python 版本
uninstall	卸载指定的 Python 版本
rehash	重新加载 Pyenv 的 shims 路径（安装完 Python 版本后需执行该命令）
version	展示当前 Python 版本号及其生效的路径
versions	列出 Pyenv 管控的所有可用 Python 版本
which	列出要使用命令的绝对路径
whence	列出后缀命令的所有可用版本

至此，Pyenv 介绍完毕，接下来再介绍一款 Python 多管理工具 Virtualenv，它不是通过多版本管理的方式来实现系统同时兼容多 Python 环境。Virtualenv 是底层基于 Python 开发的 Python 环境隔离工具，其通过虚拟目录的方式来实现多环境的并存。其工作原理很简单：在你所需的地方创建工作目录，该目录类似系统安装的 Python 目录，保留完整的 Python 环境、解释器、标准库和第三方库等，当我们需要时，切换环境变量激活即可使用。接下来我们进一步学习 Virtualenv 的安装部署及版本管理。

1.8.2 Virtualenv 的部署与使用

Python 的第三方包成千上万，在一个 Python 环境下开发时间越久、安装依赖越多，就越容易出现依赖包冲突的问题。为了解决这个问题，开发者们开发出了 Virtualenv，它可以搭建虚拟且独立的 Python 环境。这样就可以使每个项目环境与其他项目独立开来，保持环境的干净，避免包冲突问题。另外，在开发 Python 应用程序的时候，所有第三方的包都会被 PIP 安装到系统 Python 版本的 site-packages 目录下。但如果我们要同时开发多个应用程序，那这些应用程序会共用一个 Python，这意味着所有的包都安装在系统的 Python 目录下，这不仅影响我们的正常开发工作，还有可能因为随意变更系统 Python 版本信息而造成系统的不稳定。这种情况下，每个应用可能需要各自拥有一套“独立”的 Python 运行环境。Virtualenv 就是用来为一个应用创建一套“隔离”的 Python 运行环境的。下面我们来看看 Virtualenv 的部署，以及它如何管理 Python 环境。

(1) 部署

假设你已经学习过我们上节内容并安装好 PIP 了，那么 Virtualenv 的安装非常简单，操作如下：

```
// 安装 virtualenv
pip install virtualenv
```

返回如下结果表示安装成功：

```
Installing collected packages: virtualenv
Successfully installed virtualenv-15.0.3
```

(2) 通过 Virtualenv 管理多 Python 版本

需强调说明的是：Virtualenv 不是通过多版本管理的方式来实现系统同时兼容多 Python 环境的，而是其通过在工作目录中虚拟完整的 Python 环境来实现 Python 多环境并存。接下来我们看 Virtualenv 的使用方式。

Virtualenv 命令的使用格式如下：

```
virtualenv [OPTIONS] DEST_DIR
```

中括号 OPTIONS 表示参数选项，是可选项，即可有可无；DEST_DIR 表示命令要执行的目录，如：

```
// 创建 /data/magedu/ 的虚拟目录
virtualenv /data/magedu/
```

可用的 OPTIONS 选项如下：

```
--version 显示当前版本号。
-h, --help 显示帮助信息。
-v, --verbose 显示详细信息。
-q, --quiet 不显示详细信息。
```


-p PYTHON_EXE, --python=PYTHON_EXE 指定所用的 python 解析器的版本, 比如 --python=python2.5 就使用 2.5 版本的解析器创建新的隔离环境。默认使用的是当前系统安装 (/usr/bin/python) 的 python 解析器。

--clear 清空非 root 用户的安装, 并从头开始创建隔离环境。

--no-site-packages 令隔离环境不能访问系统全局的 site-packages 目录。

--system-site-packages 令隔离环境可以访问系统全局的 site-packages 目录。

--unzip-setuptools 安装时解压 Setuptools 或 Distribute。

--relocatable 重定位某个已存在的隔离环境。使用该选项将修正脚本, 并令所有 .pth 文件使用相应路径。

--distribute 使用 Distribute 代替 Setuptools, 也可设置环境变量 VIRTUALENV_DISTRIBUTE 达到同样效果。

--extra-search-dir=SEARCH_DIRS 用于查找 setuptools/distribute/pip 发布包的目录。可以添加任意数量的 -extra-search-dir 路径。

--never-download 禁止从网上下载任何数据。此时, 如果在本地搜索发布包失败, virtualenv 就会报错。

--prompt==PROMPT 定义隔离环境的命令行前缀。

下面详细看看 virtualenv 在工作中的应用方式。我们先创建一个 /data/datafile/software/virtualpy/ 的虚拟工作目录, 而后再切换至虚拟环境。

// 创建虚拟工作目录

```
virtualenv /data/datafile/software/virtualpy/
```

// 通过 source 加载环境变量, 使本地环境切换至虚拟工作目录

```
source /data/datafile/software/virtualpy/bin/activate
```

看到如图 1-4 所示的 Virtualenv 虚拟工作目录标识, 表示已切换至虚拟工作目录。

```
[root@linux1st ~]# source /data/datafile/software/virtualpy/bin/activate
(virtualpy)[root@linux1st virtualpy]#
(virtualpy)[root@linux1st virtualpy]#
(virtualpy)[root@linux1st virtualpy]#
(virtualpy)[root@linux1st virtualpy]#
(virtualpy)[root@linux1st virtualpy]#
```

图 1-4 Virtualenv 虚拟工作目录

退出虚拟环境命令如下:

// 退出虚拟环境

```
Deactivate
```

看到如图 1-5 所示的退出虚拟工作目录显示正常的 BASH Shell 提示符, 表示即已退出虚拟工作目录。

```
(virtualpy)[root@linux1st virtualpy]# deactivate
[root@linux1st ~]#
[root@linux1st ~]#
```

图 1-5 退出虚拟工作目录

至此, 多版本 Python 环境管理工具 Pyenv 和 Virtualenv 介绍完毕。如果基于系统默认 Python 版本安装有问题, 可尝试基于 Pyenv 或 Virtualenv 切换 Python 版本后, 再次重试 1.7

节 Ansible 的安装步骤。

1.9 本章小结

Ansible 是运维自动化工具的后起之秀。本章前半部分我们学习了 Ansible 是什么，底层通信发展史，Ansible 发展历程等概念性知识。后半部分我们详细介绍了 Ansible 安装部署方式，同时考虑本地复杂环境可能导致的部署问题，本章后半部分我们也引申介绍了 Python 多环境扩展管理，以方便大家应对部署过程中可能出现的各类问题。当然，Ansible 的部署总体非常简单，一般出问题多数是因为系统的 glibc、gcc 等开发环境没有安装或不完整导致。在学习过程中还请严格参考本章安装操作步骤循序渐进，相信大家学完本章也有所体会。

Ansible 基础元素介绍

第1章介绍了 Ansible 的功能作用、通信发展史、基础的安装部署及处理 Ansible 安装问题所需的 Python 多环境管理工具 Pyenv 和 Virtualenv。在前期基本工作准备妥当的基础上，本章进一步深入学习 Ansible 的基础元素，会相继接触 Ansible 目录结构简介、Ansible 系列命令、Ansible Inventory 配置规范、Ansible 模式匹配规则等，其中部分内容，诸如 Inventory、Ansible-playbook 等在后续涉及章节会更深入介绍。本章主要是为大家呈现 Ansible 及系列命令的基础入门介绍，所介绍的内容相互之间没有紧密关系，可选择性地阅读感兴趣章节。

2.1 Ansible 目录结构介绍

Ansible 是开源工具，整个开发过程或二次开发均遵循 GPL 协议，所以所有源码均可见。作为一款日常工作所需的核心软件，我们有必要知道其目录分布及各目录功能。通过如下命令我们可以获取 Ansible 所有文件存放目录：

```
# rpm -ql ansible
```

该命令输出内容较多，大致分为如下几类：

- ❑ 配置文件目录 `/etc/ansible/`
- ❑ 执行文件目录 `/usr/bin/`
- ❑ Lib 库依赖目录 `/usr/lib/pythonX.X/site-packages/ansible/`
- ❑ Help 文档目录 `/usr/share/doc/ansible-X.X.X/`
- ❑ Man 文档目录 `/usr/share/man/man1/`

整体的目录概要可参考如图 2-1 所示的 Ansible 目录树结构。

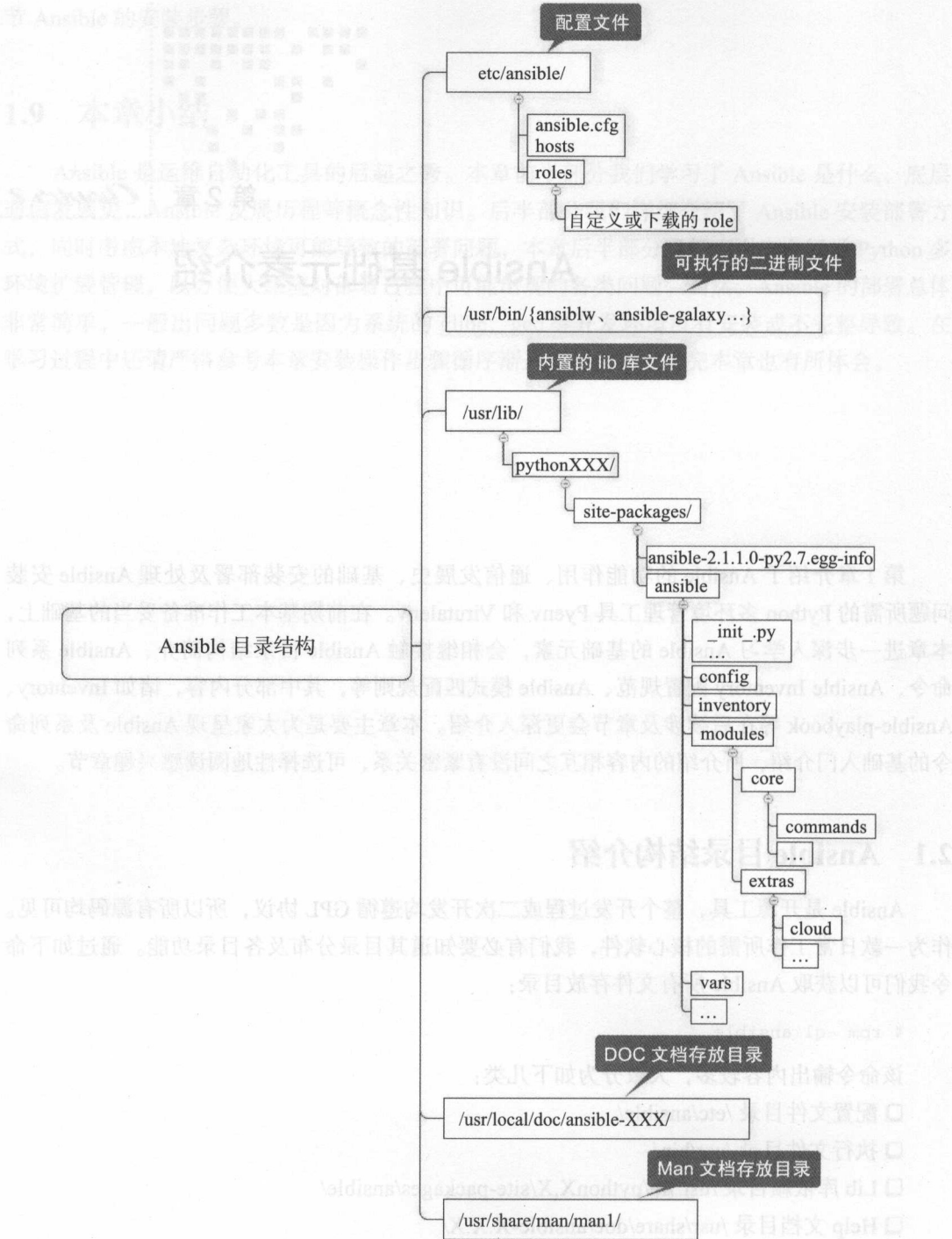


图 2-1 Ansible 目录树结构

其中,如下目录运维常要配置,需熟练掌握。

1) 配置文件目录 `/etc/ansible/`, 主要功能为: Inventory 主机信息配置、Ansible 工具功能配置等。所有 Ansible 的配置均存放在该目录下, 运维日常的所有配置类操作也均基于此目录进行。

2) 执行文件目录 `/usr/bin/`, 主要功能为: Ansible 系列命令默认存放目录。Ansible 所有的可执行文件均存放在该目录下。

在 `/usr/lib/pythonXXX/site-packages/` 下, 该目录是系统当前默认的 Python 路径, 因为 Ansible 是基于 Python 编写的, 所以 Ansible 的所有 lib 库文件和模块文件也均存放于该目录下。希望了解 Ansible 源码的话可至该目录下查看其工作原理, 当然也可至 GitHub^① 上下载历史或最新 Ansible^② 版本。

2.2 Ansible 配置文件解析

Inventory 用于定义 Ansible 的主机列表配置, Ansible 的自身配置文件只有一个, 即 `ansible.cfg`, Ansible 安装好后它默认存放于 `/etc/ansible/` 目录下。`ansible.cfg` 配置文件可以存在于多个地方, Ansible 读取配置文件的顺序依次是当前命令执行目录→用户家目录下的 `.ansible.cfg` → `/etc/ansible.cfg`, 先找到哪个就使用哪个的配置。其 `ansible.cfg` 配置的所有内容均可在命令行通过参数的形式传递或定义在 Playbooks 中。

配置文件 `ansible.cfg` 约有 350 行语句, 大多数为注释行默认配置项。该文件遵循 INI 格式, 分为如下几类配置。

(1) [defaults]

该类配置下定义常规的连接类配置, 如 `inventory`、`library`、`remote_tmp`、`local_tmp`、`forks`、`poll_interval`、`sudo_user`、`ask_sudo_pass`、`ask_pass`、`transport`、`remote_port` 等。

```
[defaults]
# inventory = /etc/ansible/hosts          # 定义 Inventory
# library = /usr/share/my_modules/         # 自定义 lib 库存放目录
# remote_tmp = $HOME/.ansible/tmp          # 临时文件远程主机存放目录
# local_tmp = $HOME/.ansible/tmp           # 临时文件本地存放目录
# forks = 5                                # 默认开启的并发数
# poll_interval = 15                       # 默认轮询时间间隔
# sudo_user = root                         # 默认 sudo 用户
# ask_sudo_pass = True                     # 是否需要 sudo 密码
# ask_pass = True                          # 是否需要密码
# roles_path = /etc/ansible/roles          # 默认下载的 Roles 存放的目录
# host_key_checking = False                # 首次连接是否需要检查 key 认证, 建议设为 False
```

① GitHub: GitHub 是一个通过 Git 进行版本控制的软件源代码托管服务, 由 GitHub 公司 (曾称 Logical Awesome) 的开发者 Chris Wanstrath、PJ Hyett 和 Tom Preston-Werner 使用 Ruby on Rails 编写而成。

② Ansible GitHub 地址: <https://github.com/ansible/ansible>。

```

# timeout = 10 # 默认超时时间
# timeout = 10 # 如没有指定用户，默认使用的远程连接用户
# log_path = /var/log/ansible.log # 执行日志存放目录
# module_name = command # 默认执行的模块
# action_plugins = /usr/share/ansible/plugins/action # action 插件的存放目录
# callback_plugins = /usr/share/ansible/plugins/callback # callback 插件的存放目录
# connection_plugins = /usr/share/ansible/plugins/connection # connection 插件的
# 存放目录
# lookup_plugins = /usr/share/ansible/plugins/lookup # lookup 插件的存放目录
# vars_plugins = /usr/share/ansible/plugins/vars # vars 插件的存放目录
# filter_plugins = /usr/share/ansible/plugins/filter # filter 插件的存放目录
# test_plugins = /usr/share/ansible/plugins/test # test 插件的存放目录
# strategy_plugins = /usr/share/ansible/plugins/strategy # strategy 插件的存放目录
# fact_caching = memory # getfact 缓存的主机信息存放方式
# retry_files_enabled = False
# retry_files_save_path = ~/.ansible-retry # 错误重启文件存放目录
...

```

上述是日常可能用到的配置，这些多数保持默认即可。

(2) [privilege_escalation]

出于安全角度考虑，部分公司不希望直接以 root 的高级管理员权限直接部署应用，往往会开放普通用户权限并给予 sudo 的权限，该部分配置主要针对 sudo 用户提权的配置。

```

[privilege_escalation]
# become=True # 是否 sudo
# become_method=sudo # sudo 方式
# become_user=root # sudo 后变为 root 用户
# become_ask_pass=False # sudo 后是否验证密码

```

(3) [paramiko_connection]

定义 paramiko_connection 配置，该部分功能不常用，了解即可。

```

[paramiko_connection] # 该配置不常用到
# record_host_keys=False # 不记录新主机的 key 以提升效率
# pty=False # 禁用 sudo 功能

```

(4) [ssh_connection]

Ansible 默认使用 SSH 协议连接对端主机，该部署是主要是 SSH 连接的一些配置，但配置项较少，多数默认即可。

```

[ssh_connection]
# pipelining = False # 管道加速功能，需配合 requiretty 使用方可生效

```

(5) [accelerate]

Ansible 连接加速相关配置。因为有部分使用者不满意 Ansible 的执行速度，所以 Ansible 在连接和执行速度方面也在不断地进行优化，该配置项在提升 Ansible 连接速度时会涉及，多数保持默认即可。

```
[accelerate]
# accelerate_port = 5099           # 加速连接端口
# accelerate_timeout = 30          # 命令执行超时时间, 单位秒
# accelerate_connect_timeout = 5.0 # 连接超时时间, 单位秒
# accelerate_daemon_timeout = 30   # 上一个活动连接的时间, 单位分钟
# accelerate_multi_key = yes
```

(6) [selinux]

关于 selinux 的相关配置几乎不会涉及, 保持默认配置即可。

```
[selinux]
# libvirt_lxc_noseclabel = yes
# libvirt_lxc_noseclabel = yes
```

(7) [colors]

Ansible 对于输出结果的颜色也进行了详尽的定义且可配置, 该选项对日常功能应用影响不大, 几乎不用修改, 保持默认即可。

```
[colors]
# highlight = white
# verbose = blue
# warn = bright purple
# error = red
# debug = dark gray
# deprecate = purple
# skip = cyan
# unreachable = red
# ok = green
# changed = yellow
# diff_add = green
# diff_remove = red
# diff_lines = cyan
```

上面尽可能全地介绍了运维工作中可能需要修改的配置选项, 除了在关闭首次连接提示 (host_key_checking = False) 或提速调整 ([accelerate] 区域块配置调整) 时可能会稍做调整, 其中绝大多数选项默认即可, Ansible 安装好后无需任何改动即可使用。

2.3 Ansible 命令用法详解

Ansible 命令行执行方式有 Ad-Hoc、Ansible-playbook 两种方式, Web 化执行方式其官方提供了付费产品 Tower (10 台以内免费), 个人的话可以基于其提供的 API 开发类似的 Web 化产品。关于命令行执行的两种方式 Ad-Hoc 和 Ansible-playbooks、什么是 Ad-Hoc 及 Ad-Hoc 与 Ansible-playbook 的区别我们在第 3 章有详细介绍, 这里不再赘述。需简要说明的是两者没有本质上的区别, Ad-Hoc 主要用于临时命令的执行, Ansibel-playbook 可以理解

为 Ad-Hoc 的集合，通过一定的规则编排在一起。两者的操作也极其简便，且提供了如 `with_items`、`failed_when`、`changed_when`、`until`、`ignore_errors` 等丰富的逻辑条件和 Dry-run 的 Check Mode。但在 Check Mode 下并不真正执行命令，即将执行的操作不会对端服务器产生任何影响，只模拟命令的执行过程是否能正常执行。

通过第 1 章的学习我们知道，Ansible 的通信默认基于 SSH，因此我们需要对主机先进行认证。Ansible 认证方式有密码认证和公私钥认证两种方式，其实完全等同于 SSH 的认证，所以这里关于这两种认证方式不做过多介绍。Ansible 默认使用（笔者也建议各位使用）公私钥认证方式，究其原因无非是出于安全的考虑，密码不用明文存放。以本机为例，执行如下命令即可添加本机认证信息。

```
// 随机生成公私钥对，ssh-keygen 是 Linux 下认证密钥生成、管理和转换工具，详细用法可参考其 man 文档
ssh-keygen -N "" -b 4096 -t rsa -C "stanley@magedu.com" -f /root/.ssh/stanley.rsa
// 为本机添加密钥认证
ssh-copy-id -i /root/.ssh/stanley.rsa root@localhost
```

输入的时候会有如下提示：

```
Are you sure you want to continue connecting (yes/no)?
```

输入全小写的英文字母 `yes` 即可。

而后会有类似如下提示输入对应 root 用户的密码信息：

```
root@localhost's password:
```

输入正确的密码信息后结果认证，随后在当前命令行输入如下命令尝试免密码登录：

```
ssh -i /root/.ssh/stanley.rsa root@localhost
```

如不提示输入密码即可直接登录则表示密钥验证成功。当然，这里只是为大家简要演示密钥认证的过程，实际应用中为方便起见，一般会使用非 root 用户生成默认文件名为 `id_rsa`、`id_rsa.pub` 的密钥对，在使用时通过 `sudo` 的方式获取权限。

Ansible 的命令使用格式如下：

```
ansible <host-pattern> [options]
```

`<host-pattern>` 是 Inventory 中定义的主机或主机组，可以为 `ip`、`hostname`、Inventory 中的 `group` 组名、具有 “.” 或 “*” 或 “:” 等特殊字符的匹配型字符串，`<>` 表示该选项是必须项，不可忽略。

`[options]` 是 Ansible 的参数选项，`[]` 表示该选项中的参数任选其一。

Ansible 命令可用选项非常多，这里列举如下会用到的选项，详细选项可参考 man 或第 3 章。

❑ `-m NAME`, `--module-name=NAME`：指定执行使用的模块。

❑ `-u USERNAME`, `--user=USERNAME`：指定远程主机以 USERNAME 运行命令。

□ `-s, --sudo`: 相当于 Linux 系统下的 `sudo` 命令。

□ `-U SUDO_USERNAME, --sudo-user=SUDO_USERNAME`: 使用 `sudo`, 相当于 Linux 下的 `sudo` 命令。

具体示例如下:

```
// 以 bruce 用户执行 ping 存活检测
ansible all -m ping -u bruce
// 以 bruce sudo 至 root 执行 ping 存活检测
ansible all -m ping -u bruce --sudo
// 以 bruce sudo 至 batman 用户执行 ping 存活检测
ansible all -m ping -u bruce --sudo --sudo-user batman
```

但在新版本中 Ansible 的 `sudo` 命令废弃, 改为 `--become` 或 `-b`, 如上命令需改为如下:

```
// 以 bruce sudo 至 root 执行 ping 存活检测
ansible all -m ping -u bruce -b
// 以 bruce sudo 至 batman 用户执行 ping 存活检测
ansible all -m ping -u bruce -b --become-user batman
```

Ansible-playbook 的命令用法和 Ansible 略有不同, 虽然参数选项与 Ansible 有很多相同的地方, 但也新增了针对 Ansible-playbook 特有的参数。

Ansible-playbook 的命令使用格式如下:

```
ansible-playbook playbook.yml
```

`ansible-playbook` 命令后跟事先编辑好的 `playbook.yml` 文件即可。本节只简单介绍其用法, 在第 4、5、6 章有大量内容介绍其写法、高级用法及优化方向。Ansible-playbook 新增的功能参数如下:

□ `--ask-vault-pass`: 加密 `playbook` 文件时提示输入密码。

□ `-D, --diff`: 当更新的文件数及内容较少时, 该选项可显示这些文件不同的地方, 该选项结合 `-C` 用会有较好的效果。

□ `-e EXTRA_VARS, --extra-vars=EXTRA_VARS`: 在 `Playbook` 中引入外部变量。

□ `--flush-cache`: 将 `fact` 清除到的远程主机缓存。

□ `--force-handlers`: 强制运行 `handlers` 的任务, 即使在任务失败的情况下。

□ `-i INVENTORY`: 指定要读取的 `Inventory` 文件。

□ `--list-tags`: 列出所有可用的 `tags`。

□ `--list-tasks`: 列出所有即将被执行的任务。

□ `--skip-tags=SKIP_TAGS`: 跳过指定的 `tags` 任务。

□ `--start-at-task=START_AT_TASK`: 从第几条任务开始执行。

□ `--step`: 逐步执行 `Playbook` 定义的任务, 并经人工确认后继续执行下一步任务。

□ `--syntax-check`: 检查 `Playbook` 中的语法书写。

□ `-t TAGS, --tags=TAGS`: 指定执行该 `tags` 的任务。

在日常工作中，大家经常会遇到批量添加认证的问题，而逐条添加认证方式，如机械地对每台主机都一条条添加认证，那是一件非常麻烦的事，也违反了我们期望的自动化方式。具体的批量添加认证方式请参考第9章。如前面所介绍，Ansible 不仅有 `ansible`、`ansible-playbook` 命令，还有 `ansible-galaxy`、`ansible-pull`、`ansible-doc`、`ansible-vault`、`ansible-console` (2.0 版本新增) 命令，掌握 Ansible 的基础用法后，我们将更深一步学习 Ansible 系列命令。

2.4 Ansible 系列命令用法详解与使用场景介绍

如何获取 Ansible 的系列命令呢？在终端键入 `ansible` 后连续按两次 Tab 键，会补全所有以 `ansible` 字母开头的命令，这些命令均是 Ansible 系列命令。本节我们来逐一介绍 Ansible 的系列命令使用。

- ❑ `ansible`
- ❑ `ansible-galaxy`
- ❑ `ansible-pull`
- ❑ `ansible-doc`
- ❑ `ansible-playbook`
- ❑ `ansible-vault`
- ❑ `ansible-console`

2.4.1 `ansible`

命令 `ansible` 是日常工作中使用率非常高的命令之一，man 中是如此定义其功能的：run a command somewhere else，可见其灵活性。`ansible` 命令主要在如下场景使用：

- ❑ 非固化需求
- ❑ 临时一次性操作
- ❑ 二次开发接口调用

那么什么是非固化需求和临时一次性操作呢？简单来讲，比如工作中我临时想查看 `web1` 服务器组是否存活，或我想临时复制本地的 `/etc/fstab` 到 `web` 服务器组的 `/tmp` 目录下做测试，类如这些没有规律的、临时需要做的任务，我们称之为非固化需求、临时一次性操作。具体的命令使用如下：

```
// 检查服务器存活
ansible web1 -m ping
// 复制本地文件到远程
ansible web1 -m copy -a "src=/etc/fstab dest=/tmp/fstab owner=root group=root
mode=644 backup=yes"
```

Ansible 的返回结果都非常友好，一般会用 3 种颜色来表示执行结果：红色、绿色、橘黄色。其中红色表示执行过程有异常，一般会中止剩余所有的任务，如图 2-2 所示的 Ansible 执行结果错误的结果返回；绿色和橘黄色表示执行过程没有异常，所有任务均正常执行，但橘黄色表示命令执行结束后目标有状态的变化。如图 2-3 所示，Ansible 执行结果正确的结果返回中的圆圈 1 为橘黄色显示；而绿色表示命令执行结束后目标没有状态变化，如图 2-3 中的圆圈 2 显示。

```
[%7%root@DS128 ~]# ansible web2 -m ping
```

图 2-2 Ansible 执行结果错误的结果返回

```
[%4%root@DS128 ~]# ansible localhost -m copy -a "src=/etc/fstab dest=/tmp/fstab"
localhost | SUCCESS => {
  "changed": true,
  "checksum": "1f831af9d44d3257ccff12d08541aea487f2a64c",
  "dest": "/tmp/fstab",
  "gid": 0,
  "group": "root",
  "md5sum": "aaaa8f18d7647d5b02ce734592f83481",
  "mode": "0644",
  "owner": "root",
  "secontext": "unconfined_u:object_r:admin_home_t:s0",
  "size": 595,
  "src": "/root/.ansible/tmp/ansible-tmp-1471863715.22-58995077405044/source",
  "state": "file",
  "uid": 0
}

[%5%root@DS128 ~]# ansible localhost -m copy -a "src=/etc/fstab dest=/tmp/fstab"
localhost | SUCCESS => {
  "changed": false,
  "checksum": "1f831af9d44d3257ccff12d08541aea487f2a64c",
  "dest": "/tmp/fstab",
  "gid": 0,
  "group": "root",
  "mode": "0644",
  "owner": "root",
  "path": "/tmp/fstab",
  "secontext": "unconfined_u:object_r:admin_home_t:s0",
  "size": 595,
  "state": "file",
  "uid": 0
}
```

图 2-3 Ansible 执行结果正确的结果返回

不仅 ansible 命令的执行结果如此设置，Ansible 系列命令均如此设置，所以判断 Ansible 系列命令的执行结果是否正常是一件非常容易的事情，只要看颜色即可。

2.4.2 ansible-galaxy

这里的 galaxy 和三星手机没有任何关系。ansible-galaxy 的功能可以简单地理解为 GitHub 或 PIP 的功能，通过 ansible-galaxy 命令，我们可以根据下载量和关注量等信息，查找和安装优秀的 Roles。Roles 是 Ansible 非常重要的一项功能，关于 Roles 的详细功能会在第 6 章介绍。在 ansible-galaxy 上，我们可以上传和下载 Roles，这里也是优秀 Roles 的聚集地，下载地址为 <https://galaxy.ansible.com>。

ansible-galaxy 命令使用格式如下：

```
ansible-galaxy [init|info|install|list|remove] [--help] [options] ...
```

ansible-galaxy 命令分三大部分:

(1) [init|info|install|list|remove]

init: 初始化本地的 Roles 配置, 以备上传 Roles 至 galaxy。

□ info: 列表指定 Role 的详细信息。

□ install: 下载并安装 galaxy 指定的 Roles 到本地。

□ list: 列出本地已下载的 Roles。

□ remove: 删除本地已下载的 Roles。

Ansible 2.0 版本中, 针对 ansible-galaxy 增加了 login、import、delete、setup 等功能, 但这些功能需基于 login 在 galaxy 认证成功后方可执行, 主要为了方便对 galaxy 上已有的 Roles 的配置工作。

(2) help 用法显示 [--help]

针对第一部分的 init、info 等功能, 其后跟 --help 可单独显示该项用法。例如:

```
ansible-galaxy init --help
```

执行后会返回 ansible-galaxy init 选项的用法说明。

```
Usage: ansible-galaxy init [options] role_name
```

Options:

-f, --force	Force overwriting an existing role
-h, --help	show this help message and exit
-c, --ignore-certs	Ignore SSL certificate validation errors.
-p INIT_PATH, --init-path=INIT_PATH	The path in which the skeleton role will be created. The default is the current working directory.
--offline	Don't query the galaxy API when creating roles
-s API_SERVER, --server=API_SERVER	The API server destination
-v, --verbose	verbose mode (-vvv for more, -vvvv to enable connection debugging)
--version	show program's version number and exit

其他选项与 help 用法一样。

(3) 参数项 [options]

该部分结合第一部分的参数完成 ansible-galaxy 完整的功能用法, 如:

ansible-galaxy init [options] role_name 即 ansible-galaxy init 后跟 [-f|-h|-c|-p|--offline|-s SERVER|-v|--version] 参数, 后跟 role-name 成为一条完整的命令。

具体可参考如下:

```
// 下载用户 hectcastro 的 Nginx 这个 Role 到本地并忽略错误 (默认存放在 /etc/ansible/roles/)
ansible-galaxy --ignore-errors install azavea.git
```


因为 ansible-galaxy 是对 <https://galaxy.ansible.com> 网站的上传、下载、配置类工作，如有类似如下报错，请确保该网站可正常访问。

```
the API server (galaxy.ansible.com) is not responding, please try again later.
```

2.4.3 ansible-pull

该指令的使用涉及 Ansible 的另一种工作模式：pull 模式（Ansible 默认使用 push 模式）。这和通常使用的 push 模式工作机理刚好相反，其适用于以下场景：①你有数量巨大的机器需要配置，即使使用高并发线程依旧要花费很多时间；②你要在刚启动的、没有网络连接的主机上运行 Ansible。

ansible-pull 命令使用格式如下：

```
ansible-pull [options] [playbook.yml]
```

通过 ansible-pull 结合 Git 和 crontab 一并实现，其原理如下：通过 crontab 定期拉取指定的 Git 版本到本地，并以指定模式自动运行预先制订好的指令。

具体示例参考如下：

```
*/20 * * * * root /usr/local/bin/ansible-pull -o -C 2.1.0 -d /srv/www/king-gw/
-i /etc/ansible/hosts -U git://git.kingifa.com/king-gw-ansiblepull >> /var/log/
ansible-pull.log 2>&1
```

ansible-pull 通常在配置大批量机器的场景下会使用，灵活性稍有欠缺，但效率几乎可以无限提升，对运维人员的技术水平和前瞻性规划有较高要求。

2.4.4 ansible-doc

ansible-doc 是 Ansible 模块文档说明，针对每个模块都有详细的用法说明及应用案例介绍，功能和 Linux 系统 man 命令类似。该命令使用方式如下：

```
ansible-doc [options] [module...]
```

ansible-doc 命令后跟 [options] 参数或 [模块名]，显示模块用法说明，具体示例如下：

```
// 列出支持的模块
ansible-doc -l
// 模块功能说明
ansible-doc ping
```

2.4.5 ansible-playbook

ansible-playbook 是日常应用中使用频率最高的命令，其工作机制是：通过读取预先编写好的 playbook 文件实现批量管理。要实现的功能与命令 ansible 一样，可以理解为按一定条件组成的 ansible 任务集。

ansible-playbook 命令后跟 YML 格式的 playbook 文件，执行事先编排好的任务集，命令

使用方式如下：

```
ansible-playbook playbook.yml
```

具体示例如下：

```
// 执行 gw.yml 这个 playbook 中定义的所有任务集
ansible-playbook gw.yml
```

Playbook 具有编写简单、可定制性高、灵活方便，以及可固化日常所有操作的特点，运维人员应熟练掌握。

2.4.6 ansible-vault

ansible-vault 主要用于配置文件加密，如编写的 Playbook 配置文件中包含敏感信息，不希望其他人随意查看，ansible-vault 可加密/解密这个配置文件，具体使用方式如下：

```
Usage: ansible-vault [create|decrypt|edit|encrypt|rekey|view] [--help] [options]
file_name
```

具体示例如下。

设定如下密码，加密 a.yml 文件。

```
ansible-vault encrypt a.yml
```

会有以下输入加密密码提示：

```
Vault password:
Confirm Vault password:
Encryption successful
```

这时，再打开 a.yml 文件后会发现该文件乱码，只有通过如下命令解密后方可正常查看。

```
ansible-vault decrypt a.yml
```

输入预设的密码后方可解密。

```
Vault password:
Decryption successful
```

此时 a.yml 文件可正常查看。

到此，我们对 Ansible 的用法及系列命令已经有了概念性的了解和掌握，接下来我们进一步了解 Ansible Inventory 文件的配置管理。

2.4.7 ansible-console

ansible-console 是 Ansible 为用户提供的一款交互式工具，用户可以在 ansible-console 虚拟出来的终端上像 Shell 一样使用 Ansible 内置的各种命令，这为习惯于使用 Shell 交互方式

的用户提供了良好的使用体验。ansible-console 主要是针对 1.X 版本中 ansible-shell 工具而研发的, 目前官网还没有对 ansible-console 进行详细的用法说明, 最新版本的 Ansible 软件包也没有对应的 man 文档说明。经笔者实测, ansible-console 命令的使用格式如下。

在终端键入 ansible-console 命令后, 会进入如图 2-4 所示的类似 Shell 一样的交互式终端环境。

```
[146%root@DS128 /usr/local/share/man]# ansible-console
welcome to the ansible console.
Type help or ? to list commands.

root@all (4)[f:5]$ ?      列出所有的可用模块

Documented commands (type help <topic>):
=====
EOF                        nagios
a10                        netcaler
a10_server                 network
a10_service_group          newrelic_deployment
a10_virtual_server         nexmo
accelerate                 nmcli
acl                         notification
add_host                   nova_compute
```

图 2-4 ansible-console 命令用法 1

图 2-4 中的 “root@all (4)[f:5]\$” 是提示符, 该提示符表示 “当前的使用用户 @ 当前所在的 Inventory 中定义的组, 默认是 all 分组 (Inventory 中 all 组所有主机的数量) [forks: 线程数 \$]”。

使用 cd 命令可切换至指定 Hosts 或分组, 同时提示符的相应信息也会随之变动, 如图 2-5 所示。

```
root@all (4)[f:5]$ cd webs      当前webs分组主机数量
root@webs (3)[f:5]$ forks 2    设置并发线程数为2个
root@webs (3)[f:2]$ list
192.168.99.136
192.168.99.150
192.168.99.151                webs分组的所有主机列表
```

图 2-5 ansible-console 命令用法 2

如图 2-5 中的 Ansible-console 命令用法 2 所示, cd 至 webs 分组后, 原来的 root@all (4) [f:5]\$ 也相应地变更为 root@webs (3)[f:5], 表示当前分组为 webs 分组, 该分组所拥有的主机总数为 3 台。执行 forks 2 后, 提示符再次变更为 root@webs (3)[f:2]\$, 表示设置并发的线程数为 2。

所有的操作与 Shell 类似, 而且支持 Tab 键补全, 如启动 httpd 服务时, 键入 service 后连续按两次 Tab 键后会自动补全剩余的命令选项。ansible-console 命令用法 3 如图 2-6 所示。

```
root@webs (3)[f:2]$ service
arguments= name=          runlevel=  state=
enabled=   pattern=      sleep=
root@webs (3)[f:2]$ service name=httpd state=started
```

图 2-6 ansible-console 命令用法 3

如需启动 httpd 服务, 使用命令 `service name=httpd state=started`, 命令用法与 Ad-Hoc 一致, 只是格式上的使用习惯不同而已。如想获取 service 模块更详细的用法, 输入 `help service` 命令即可。ansible-console 命令用法 4 如图 2-7 所示。

```
root@webs (3)[f:2]$ help service
Manage services.
Parameters:
  state C(started)/C(stopped) are idempotent actions that will not run commands un
less necessary. C(restarted) will always bounce the service. C(reloaded) will al
ways reload. B(At least one of state and enabled are required.)
  sleep If the service is being C(restarted) then sleep this many seconds between
the stop and start command. This helps to workaround badly behaving init scripts t
hat exit immediately after signaling a process to stop.
  name Name of the service.
  runlevel For OpenRC init scripts (ex: Gentoo) only. The runlevel that this serv
ice belongs to.
  pattern If the service does not respond to the status command, name a substring
to look for as would be found in the output of the I(ps) command as a stand-in for
a status result. If the string is found, the service will be assumed to be runni
ng.
  enabled whether the service should start on boot. B(At least one of state and en
abled are required.)
  arguments Additional arguments provided on the command line
```

图 2-7 ansible-console 命令用法 4

使用完毕如希望退出, 按快捷键 Ctrl+D 或 Ctrl+C 即可退出当前的虚拟终端。

ansible-console 命令在实际工作中用于 Ad-Hoc 和 Ansible-playbooks 之间的场景, 常用于集中一批临时操作或命令, 使用 Ad-Hoc 要键入很多次但整体操作的复杂度又不至于使用 Playbooks 时, 这时 ansible-console 是最佳选择。

2.5 Ansible Inventory 配置及详解

Inventory 是 Ansible 管理主机信息的配置文件, 相当于系统 HOSTS 文件的功能, 默认存放在 `/etc/ansible/hosts`。为方便批量管理主机, 便捷使用其中的主机分组, Ansible 通过 Inventory 来定义其主机和组, 在使用时通过 `-i` 或 `--inventory-file` 指定读取, 与 Ansible 命令结合使用时组合如下:

```
ansible -i /etc/ansible/hosts webs -m ping
```

如果只有一个 Inventory 时可不用指定路径, 默认读取 `/etc/ansible/hosts`。Inventory 可以同时存在多个, 而且支持动态生成, 如 AWS EC2、Cobbler 等均支持, 本节我们来学习 Inventory 的使用规则。

2.5.1 定义主机和组

Inventory 配置文件遵循 INI 文件风格, 中括号中的字符为组名。其支持将同一个主机同时归并到多个不同的组中, 分组的功能为 IT 人员维护主机列表提供了非常大的便利。此外, 若目标主机使用了非默认的 SSH 端口, 还可以在主机名称之后使用冒号加端口号来标明, 以

行为单位分隔配置，详细信息可参考以下代码中的注释。

```
# “#”开头的行表示该行为注释行，即当时行的配置不生效
# Inventory 可以直接为 IP 地址
192.168.37.149
# Inventory 同样支持 Hostname 的方式，后跟冒号加数字表示端口号，默认 22 号端口
ntp.magedu.com:2222
nfs.magedu.com
# 中括号内的内容表示一个分组的开始，紧随其后的主机均属于该组成员，空行后的主机亦属于该组，即
web2.magedu.com 这台主机也属于 [websevers] 组
[websevers]
web1.magedu.com
web[10:20].magedu.com # [10:20] 表示 10 ~ 20 之间的所有数字（包括 10 和 20），即表示 web10.
magedu.com、web11.magedu.com……web20.magedu.com 的所有主机

web2.magedu.com[dbservers]
db-a.magedu.com
db-[b:f].magedu.com # [b:f] 表示 b 到 f 之间的所有数字（包括 b 和 f），即表示 db-b.magedu.
com、db-e.magedu.com……db-f.magedu.com 的所有主机
```

2.5.2 定义主机变量

在日常工作中，通常会遇到非标准化的需求配置，如考虑到安全性问题，业务人员通常将企业内部的 Web 服务 80 端口修改为其他端口号，而该功能可以直接通过修改 Inventory 配置来实现，在定义主机时为其添加主机变量，以便在 Playbook 中使用针对某一主机的个性化要求。

```
[websevers]
web1.magedu.com http_port=808 maxRequestsPerChild=801 # 自定义 http_port 的端口号为
808，配置 maxRequestsPerChild 为 801
```

Ansible 其实支持多种方式修改或自定义变量，Inventory 是其中的一种修改方式，在第 6 章中我们会详细介绍。不管哪种修改方式，大家一定要有自己的修改规范，以便于区别管理已有配置。

2.5.3 定义组变量

Ansible 支持定义组变量，主要针对大量机器的变量定义需求，赋予指定组内所有主机在 Playbook 中可用的变量，等同于逐一给该组下的所有主机赋予同一变量。定义组变量的参考案例如下：

```
[groupservers]
web1.magedu.com
web2.magedu.com

[groupservers:vars]
ntp_server=ntp.magedu.com # 定义 groupservers 组中所有主机 ntp_server 值为 ntp.magedu.
```

```
com
nfs_server=nfs.magedu.com # 定义 groupservers 组中所有主机 nfs_server 值为 nfs.magedu.
com
```

2.5.4 定义组嵌套及组变量

Inventory 中，组还可以包含其他的组（嵌套），并且也可以向组中的主机指定变量。不过，这些变量只能在 Ansible-playbook 中使用，而 Ansible 不支持。组与组之间可以相互调用，并且可以向组中的主机指定变量。

参考示例如下：

```
[apache]
httpd1.magedu.com
httpd2.magedu.com

[nginx]
ngx1.magedu.com
ngx2.magedu.com

[webservers:children]
apache
nginx

[webservers:vars]
ntp_server=ntp.magedu.com
```

Ansible 以简单为其核心理念，上述实现在业务日常使用中并不常见，大家了解其用法即可。

2.5.5 多重变量定义

变量除了可以在 Inventory 中一并定义，也可以独立于 Inventory 文件之外单独存储到 YAML 格式的配置文件中，这些文件通常以 .yaml、.yml、.json 为后缀或者无后缀。变量通常从如下 4 个位置检索：

- ❑ Inventory 配置文件（默认 /etc/ansible/hosts）
- ❑ Playbook 中 vars 定义的区域
- ❑ Roles 中 vars 目录下的文件
- ❑ Roles 同级目录 group_vars 和 hosts_vars 目录下的文件

假如 foosball 主机同属于 raleigh 和 webservers 组，那么其变量在如下文件中设置均有效：

```
/etc/ansible/group_vars/raleigh # can optionally end in '.yaml', '.yml', or
'.json'
/etc/ansible/group_vars/webservers
/etc/ansible/host_vars/foosball
```

对于变量的读取，Ansible 遵循如上优先级顺序，因此大家设置变量时尽量沿用同一种方式，以方便维护人员管理。

2.5.6 其他 Inventory 参数列表

除了支持如上的功能外，Ansible 基于 SSH 连接 Inventory 中指定的远程主机时，还内置了很多其他参数，用于指定其交互方式，如下列举了部分重要参数：

```
ansible_ssh_host: 指定连接主机
ansible_ssh_port: 指定 SSH 连接端口，默认 22
ansible_ssh_user: 指定 SSH 连接用户
ansible_ssh_pass: 指定 SSH 连接密码
ansible_sudo_pass: 指定 SSH 连接时 sudo 密码
ansible_ssh_private_key_file: 指定特有私钥文件
...
```

其他内置参数还有数十个，这些参数均可以直接写在命令行或 Playbook 文件中，以覆盖配置文件中的定义。更多参数请参考官网^①。

2.6 Ansible 与正则

正则表达式（Patterns）是各类高级语言的必定支持的方法之一，Ansible 也不例外。其 Patterns 功能等同于正则表达式，语法使用也和正则类同，这大大便利了运维的使用。其对于 Ansible 的灵活性有着极大贡献，该功能同样支持 Ansible-playbook。其用法也非常简单。

```
ansible <pattern_goes_here> -m <module_name> -a <arguments>
```

该功能主要针对 Inventory 的主机列表使用，我们通过一些案例可以更好地了解其功能及用法。在如下示例中主要针对 webservers 进行正则匹配：

```
// 重启 webservers 组所有主机的 httpd 服务
ansible webservers -m service -a "name=httpd state=restarted"
```

(1) All (全量) 匹配

匹配所有主机，all 或 * 号功能相同。如检测所有主机存活情况。

```
// all 和 * 功能相同，但 * 号需引起来
ansible all -m ping
ansible "*" -m ping
```

检查 192.168.1.0/24 网段所有主机存活状况。

```
ansible 192.168.1.* -m ping
```

(2) 逻辑或 (or) 匹配

如我们希望同时对多台主机或多个组同时执行，相互之间用 “:” (冒号) 分隔即可。

① 内置变量参考网址：https://docs.ansible.com/ansible/intro_inventory.html#list-of-behavioral-inventory-parameters。

```
web1:web2
```

使用方式如下：

```
ansible "web1:web2" -m ping
```

(3) 逻辑非 (!) 匹配

逻辑非用感叹号 (!) 表示，主要针对多重条件的匹配规则，使用方式如下：

```
// 所有在 webserver 组但不在 phoenix 组的主机
webserver:!phoenix
```

(4) 逻辑与 (&) 匹配

和逻辑非一样，逻辑与也主要针对多重条件的匹配规则，只是逻辑上的判断不同。逻辑与使用 & 表示，请看如下示例：

```
// webserver 组和 staging 组中同时存在的主机
webserver:&staging
```

(5) 多条件组合

Ansible 同样支持多条件的复杂组合，该情况企业应用不多，这里做简单举例说明。

```
// webserver 和 dbserver 两个组中的所有主机在 staging 组中存在且在 phoenix 组中不存在的主机
webserver:dbserver:&staging:!phoenix
```

(6) 模糊匹配

* 通配符在 Ansible 表示 0 个或多个任意字符，主要应用于一些模糊规则匹配，在平时的使用中应用频率非常高，请参考如下示例：

```
// 所有以 .magedu.com 结尾的主机均符合
*.magedu.com
// one 开头 .com 结尾的所有主机和 dbserver 组中的所有主机
one*.com:dbserver
```

(7) 域切割

Ansible 底层基于 Python，因此也支持域切割。Python 字符串域切割的示例如下：

```
str = '12345678'
print str[0:1]
```

通过 [0:1] 即可获取数值 1。该功能在 Ansible 中也支持，以如下 Inventory 内容为例：

```
[webserver]
cobweb
webbing
weber
```

通过截取数组下标可以获得对应变数值。

```
webserver[0] # == cobweb
```



```

webservers[-1]      # == weber
webservers[0:1]     # == webservers[0],webservers[1]
                    # == cobweb,webbing
webservers[1:]      # == webbing,weber

```

(8) 正则匹配

Ansible 同样完整支持正则匹配功能，“~”开始表示正则匹配。

```
~(web|db).*\.example\.com
```

检测 beta.example.com、web.example.com、green.example.com、beta.example.org、web.example.org、green.example.org 的存活，使用如下匹配模式：

```
ansible "(beta|web|green)\.example\.(com|org)" -m ping
```

检测 Inventory 中所有以 192.168 开头的服务器存活信息：

```
ansible ~192\.168\.[0-9]\{\2\}\.[0-9]\{\2,\} -m ping
```

关于 Ansible 的正则功能到此结束，相信大家在浏览的过程中对其灵活程度也会有所感触，在对 Ansible 的实际应用过程中也会不断地加深对其理解。

2.7 本章小结

本章着重为大家介绍了 Ansible 目录结构功能及 Ansible 命令使用方式。在对 Ansible 的使用有概念性的了解后，又介绍 Ansible 系列命令的功能作用，通过简单的案例加深对各命令功能的理解。Inventory 是 Ansible 的核心功能点之一，本章用了约一半内容讲解了 Inventory 的入门、书写规范、使用技巧及其与正则的结合使用。正则作为运维必备技能与 Ansible 的结合也使得 Ansible 愈显灵活，功能也愈显强大，请务必熟练掌握。下章我们将为大家介绍 Ansible Ad-Hoc 命令集的进阶使用。

Ansible Ad-Hoc 命令集

第2章介绍了 Ansible 的各项元素、系列命令、Inventory 基础，以及 Ansible 与正则的结合使用，这些内容是掌握 Ansible 的基础，请务必熟练掌握。在前两章的基础上，本章为大家介绍 Ansible Ad-Hoc 命令集，通过模拟真实的企业案例和应用场景更深入地了解 Ansible。作为 Ansible 最常用的命令，本节内容显得尤为重要。

3.1 Ad-Hoc 使用场景

所谓 Ad-Hoc，简而言之是“临时命令”，英文中作为形容词有“特别的，临时”的含义。Ad-Hoc 只是官方对 Ansible 命令的一种称谓，大家按各自习惯称呼即可。笔者平时一般称之为“临时操作”或 Ansible 命令。

从功能上讲，Ad-Hoc 是相对 Ansible-playbook 而言的，Ansible 提供两种完成任务方式：一种是 Ad-Hoc 命令集，即命令 `ansible`，另外一种就是 Ansible-playbook 了，即命令 `ansible-playbook`。前者更侧重于解决一些简单或者平时工作中临时遇到的任务，相当于 Linux 系统命令行下的 Shell 命令，后者更适用于解决复杂或需固化下来的任务，相当于 Linux 系统的 Shell Scripts。通常，深入 Ansible 是从接触 Ansible-playbook 开始的，灵活运用 Ansible-playbook 才能更好地体会到 Ansible 的强大所在。

具体来讲，什么样的场景下我们需要用到 Ad-Hoc，什么样的情况下需要使用 Ansible-playbook 呢？

(1) 需要使用 Ad-Hoc 的场景

情景 1:

节假日将至，我们需要关闭所有不必要的服务器，并对所有服务器进行节前健康检查。

情景 2:

临时更新 Apache & Nginx 的配置文件，且需同时将其分发至所有需更新该配置的 Web 服务器。

(2) 需要使用 Ansible-playbook 的场景

情景 1:

新购置的服务器安装完系统后需做一系列固化的初始化工作，诸如：定制防火墙策略、添加 NTP 时间同步配置、添加 EPEL 源等。

情景 2:

业务侧每周定期对生产环境发布更新程序代码。

其实两者之间关系用急行军 (Ad-Hoc) 和远征军 (Ansible-playbook) 来形容可能更容易理解。急行军需轻装上阵，注重灵活机动；远征军需稳扎稳打，注重长远规划。正如我们上面所讲，Ad-Hoc 更侧重于解决一些简单或者平时工作中临时遇到的任务，Ansible-playbook 更适合于解决复杂的或需固化下来的任务。后面的章节中我们会介绍大量企业实战场景，相信大家会有更深刻的体会。

3.2 Ad-Hoc 命令集介绍

本节介绍通过 Ad-Hoc 命令集查看系统设置，通过 Ad-Hoc 研究 Ansible 的并发特性，通过 Ad-Hoc 研究 Ansible 的模块使用。俗话说，磨刀不误砍柴工。开始之前做一些简单的初始化检查，如系统时间正确与否、磁盘容量是否充足等，是很有必要的。



提示 在实际工作中，很多“诡异”问题迫使我们花费大量时间排查，最终却发现是非常简单的基础环境问题导致的。这其实还是挺常见的，不论对新手还是老鸟均如此，谨记！

我们前面做的系统时间正确与否、磁盘容量是否充足等工作，其实 Linux 下是有开源工具可以帮助我们自动监控的。这里也为大家推荐几款 Linux 下耳熟能详的监控工具，如 Zabbix、Nagios、Cacti、falcon、Cat 等。

3.2.1 Ad-Hoc 命令集用法简介

本节我们介绍 Ad-Hoc 命令集用法。Ad-Hoc 命令集由 /usr/bin/ansible 实现，其命令用法如下：

```
ansible <host-pattern> [options]
```

可用选项如下。

□ -v, --verbose: 输出更详细的执行过程信息，-vvv 可得到执行过程所有信息。

- ❑ `-i PATH, --inventory=PATH`: 指定 inventory 信息, 默认 `/etc/ansible/hosts`。
- ❑ `-f NUM, --forks=NUM`: 并发线程数, 默认 5 个线程。
- ❑ `--private-key=PRIVATE_KEY_FILE`: 指定密钥文件。
- ❑ `-m NAME, --module-name=NAME`: 指定执行使用的模块。
- ❑ `-M DIRECTORY, --module-path=DIRECTORY`: 指定模块存放路径, 默认 `/usr/share/ansible`, 也可以通过 `ANSIBLE_LIBRARY` 设定默认路径。
- ❑ `-a 'ARGUMENTS', --args='ARGUMENTS'`: 模块参数。
- ❑ `-k, --ask-pass SSH`: 认证密码。
- ❑ `-K, --ask-sudo-pass sudo`: 用户的密码 (`--sudo` 时使用)。
- ❑ `-o, --one-line`: 标准输出至一行。
- ❑ `-s, --sudo`: 相当于 Linux 系统下的 `sudo` 命令。
- ❑ `-t DIRECTORY, --tree=DIRECTORY`: 输出信息至 `DIRECTORY` 目录下, 结果文件以远程主机名命名。
- ❑ `-T SECONDS, --timeout=SECONDS`: 指定连接远程主机的最大超时, 单位是秒。
- ❑ `-B NUM, --background=NUM`: 后台执行命令, 超 `NUM` 秒后中止正在执行的任务。
- ❑ `-P NUM, --poll=NUM`: 定期返回后台任务进度。
- ❑ `-u USERNAME, --user=USERNAME`: 指定远程主机以 `USERNAME` 运行命令。
- ❑ `-U SUDO_USERNAME, --sudo-user=SUDO_USERNAME`: 使用 `sudo`, 相当于 Linux 下的 `sudo` 命令。
- ❑ `-c CONNECTION, --connection=CONNECTION`: 指定连接方式, 可用选项 `paramiko` (SSH)、`ssh`、`local`, `local` 方式常用于 `crontab` 和 `kickstarts`。
- ❑ `-l SUBSET, --limit=SUBSET`: 指定运行主机。
- ❑ `-l ~REGEX, --limit=~REGEX`: 指定运行主机 (正则)。
- ❑ `--list-hosts`: 列出符合条件的主机列表, 不执行任何命令。

下面的示例有助于加深对上述内容的理解。

情景 1: 检查 `proxy` 组所有主机是否存活。

执行命令:

```
ansible proxy -f 5 -m ping
```

返回结果如图 3-1 所示。

```
[root@linuxl1st ~]# ansible proxy -f 5 -m ping
192.168.37.159 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
```

图 3-1 ansibleping 命令返回结果

执行结果诠释：

```
192.168.37.159 | success >> {
    "changed": false,
    "ping": "pong"
}
```

其中 192.168.37.159 是指命令执行的主机，Success 表示命令执行成功，“>> {}”表示详细返回结果如下。“"changed": false”表示没有对主机做变更，“"ping": "pong"”表示执行了 ping 命令返回结果为 pong。

情景 2：返回 proxy 组所有主机的 hostname，并打印最详细的执行过程到标准输出。

执行命令：

```
ansible proxy -s -m command -a 'hostname' -vvv
```

返回结果如图 3-2 所示。

```
[root@linuxlst ~]# ansible proxy -s -m command -a 'hostname' -vvv
Using /etc/ansible/ansible.cfg as config file
<192.168.37.159> ESTABLISH SSH CONNECTION FOR USER: None
<192.168.37.159> SSH: EXEC ssh -C -q -o ControlMaster=auto -o ControlPersist=60s -o KbdInteractiveAuthentication=no -o PreferredAuthentications=gssapi-with-mic,gssapi-keyex,hostbased,publickey -o PasswordAuthentication=no -o ConnectTimeout=10 -o ControlPath=/root/.ansible/cp/ansible-ssh-%h-%p-%r 192.168.37.159 '/bin/sh -c "'
'(' umask 77 && mkdir -p " echo $HOME/.ansible/tmp/ansible-tmp-1469876722.79-37867079091680 " && echo ansible-tmp-1469876722.79-37867079091680=" echo $HOME/.ansible-tmp/ansible-tmp-1469876722.79-37867079091680 "' ) && sleep 0"'
<192.168.37.159> PUT /tmp/tmpZGdiAz TO /root/.ansible/tmp/ansible-tmp-1469876722.79-37867079091680/command
<192.168.37.159> SSH: EXEC sftp -b - -C -o ControlMaster=auto -o ControlPersist=60s -o KbdInteractiveAuthentication=no -o PreferredAuthentications=gssapi-with-mic,gssapi-keyex,hostbased,publickey -o PasswordAuthentication=no -o ConnectTimeout=10 -o ControlPath=/root/.ansible/cp/ansible-ssh-%h-%p-%r '[192.168.37.159]'
<192.168.37.159> ESTABLISH SSH CONNECTION FOR USER: None
<192.168.37.159> SSH: EXEC ssh -C -q -o ControlMaster=auto -o ControlPersist=60s -o KbdInteractiveAuthentication=no -o PreferredAuthentications=gssapi-with-mic,gssapi-keyex,hostbased,publickey -o PasswordAuthentication=no -o ConnectTimeout=10 -o ControlPath=/root/.ansible/cp/ansible-ssh-%h-%p-%r -tt 192.168.37.159 '/bin/sh -c "'
'""'sudo -H -S -n -u root /bin/sh -c 'echo BECOME-SUCCESS-bdskaudhnjvzsidzmfzxszyxmlxmwxvd; LANG=en_US.UTF-8 LC_ALL=en_US.UTF-8 LC_MESSAGES=en_US.UTF-8 /usr/bin/python /root/.ansible/tmp/ansible-tmp-1469876722.79-37867079091680/command; rm -rf "/root/.ansible/tmp/ansible-tmp-1469876722.79-37867079091680/" > /dev/null 2>&1' && sleep 0"'
192.168.37.159 | SUCCESS | rc=0 >>
linuxlst
```

图 3-2 ansiblehostname 命令返回结果

执行结果诠释：

```
<192.168.37.159> ESTABLISH CONNECTION FOR USER: root on PORT 22 TO
192.168.37.159 # 远程主机 192.168.37.159 监听 ROOT 用户的 22 号端口
<192.168.37.159> REMOTE_MODULE command hostname # 远程执行命令 hostname
<192.168.37.159> EXEC /bin/sh -c 'mkdir -p $HOME/.ansible/tmp/ansible-tmp-1455684626.83-94958346638443 && echo $HOME/.ansible/tmp/ansible-tmp-1455684626.83-94958346638443' # 生成临时目录用于存放 Ansible 远程执行脚本
<192.168.37.159> PUT /tmp/tmp5sawsq TO /root/.ansible/tmp/ansible-
```

```
tmp-1455684626.83-94958346638443/command # 改名临时脚本并存放至临时目录
<192.168.37.159> EXEC /bin/sh -c 'sudo -k && sudo -H -S -p "[sudo via ansible,
key=urvzacjxvaagwlvrywmxpxfhjkirkqb] password: " -u root /bin/sh -c \''''echo
BECOME-SUCCESS-urvzacjxvaagwlvrywmxpxfhjkirkqb; LANG=C LC_CTYPE=C /usr/bin/
python /root/.ansible/tmp/ansible-tmp-1455684626.83-94958346638443/command; rm
-rf /root/.ansible/tmp/ansible-tmp-1455684626.83-94958346638443/ >/dev/null
2>&1'''' # 使用 sudo 方式并以 Python 脚本方式执行命令
192.168.37.159 | success | rc=0 >> # 返回结果为 success, CodeResult 为 0
Linuxlst # 返回的命令返回结果如下
```

使用 -vvv 参数可以清楚地了解 Ansible 命令执行流程，如图 3-3 所示。

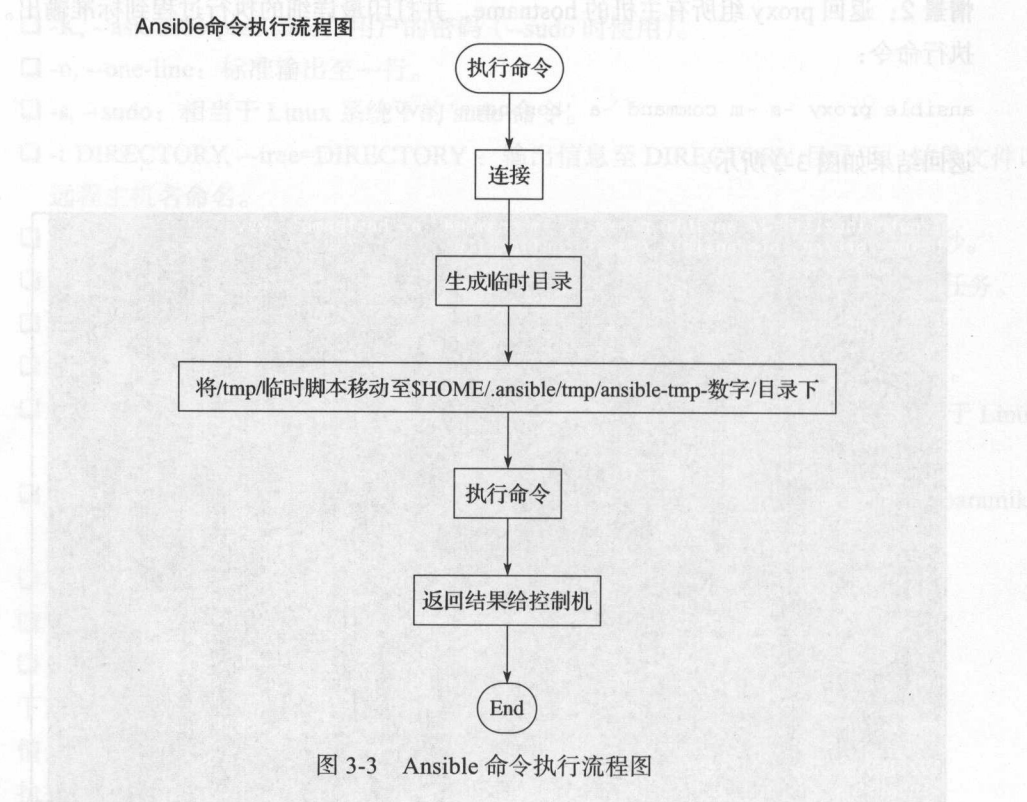


图 3-3 Ansible 命令执行流程图

情景 3：列出 Web 组所有主机列表。

执行命令：

```
ansible web --list
```

返回结果如下：

```
10.3.33.21
10.3.33.23
```

执行结果诠释：

--list 选项列出 web 组所有主机列表信息, Web 组中包括 10.3.33.21 和 10.3.33.23 两台主机

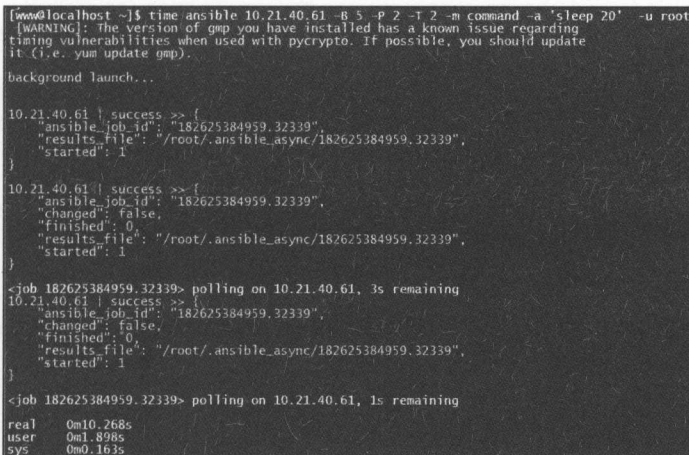
接下来我们模拟较为复杂的场景。

情景 4: 对 10.21.40.61 服务器以 root 执行 sleep 20, 设置最大连接超时时长为 2s, 且设置为后台运行模式, 执行过程每 2s 输出一次进度, 如 5s 还未执行完则终止该任务。

执行命令:

```
//time 命令可省, 为方便观察结果, 这里使用 time 命令查看执行时长
time ansible 10.21.40.61 -B 5 -P 2 -T 2 -m command -a 'sleep 20' -u root
```

返回结果如图 3-4 所示。



```
[root@localhost ~]$ time ansible 10.21.40.61 -B 5 -P 2 -T 2 -m command -a 'sleep 20' -u root
[WARNING]: The version of gmp you have installed has a known issue regarding
timing vulnerabilities when used with pycrypto. If possible, you should update
it (i.e. yum update gmp).
background launch...
10.21.40.61 | success >> {
  "ansible_job_id": "182625384959.32339",
  "results_file": "/root/.ansible_async/182625384959.32339",
  "started": 1
}
10.21.40.61 | success >> {
  "ansible_job_id": "182625384959.32339",
  "changed": false,
  "finished": 0,
  "results_file": "/root/.ansible_async/182625384959.32339",
  "started": 1
}
<job 182625384959.32339> polling on 10.21.40.61, 3s remaining
10.21.40.61 | success >> {
  "ansible_job_id": "182625384959.32339",
  "changed": false,
  "finished": 0,
  "results_file": "/root/.ansible_async/182625384959.32339",
  "started": 1
}
<job 182625384959.32339> polling on 10.21.40.61, 1s remaining
real    0m10.268s
user    0m1.898s
sys     0m0.163s
```

图 3-4 返回结果

执行结果诠释:

第 1 行: [WARNING] 不用理会, 需升级 gmp, 该提醒不影响命令返回结果

第 2 行: background launch... 表示使用 -B 使该命令后台运行

下面每隔 2s 输出一次执行进度

<job 182625384959.32339> polling on 10.21.40.61, 3s remaining 表示执行时长剩余 3s

下面每隔 2s 输出一次执行进度

<job 182625384959.32339> polling on 10.21.40.61, 1s remaining 表示执行时长剩余 1s

real 0m10.268s 程序执行总时长

user 0m1.898s 系统用户层执行时长

sys 0m0.163s 系统内核层执行时长



提示

细心的朋友会发现, 我们 sleep 20 表示暂停 20s, 即该命令最少执行时长为 20s, 但为什么 real 程序实际运行时长只有 10s 呢? 这就是 -B 选项的意义了, 如果超过其指定时间则终止正在执行的任务 (但 real 为什么是 10.268s 而不是 5.268s, 经笔者实测, -B 功能生效但时间不精确, 正式使用前请多测试)。

以上为 Ad-Hoc 命令集的用法说明，后面的章节我们会通过更复杂的实例深入了解其功能。

3.2.2 通过 Ad-Hoc 查看系统设置

3.2.1 节为大家介绍了 Ad-Hoc 的命令集用法，本节我们通过 df、free 命令查看系统设置，但是是通过 Ad-Hoc 实现的，在此过程中帮助大家了解 Ansible。我们模拟如下两个场景。

情景 1：批量查看 apps 组所有主机的磁盘容量（使用 command 模块）。

执行命令：

```
ansible apps -a "df -lh"
```

返回结果如下：

```
192.168.37.130 | success | rc=0 >>
Filesystem                Size      Used Avail Use% Mounted on
/dev/mapper/vg_linuxl1st-lv_root
19G    3.6G    14G    21% /
tmpfs                    123M        0   123M    0% /dev/shm
/dev/sda1                 485M     29M   431M    7% /boot

192.168.37.155 | success | rc=0 >>
Filesystem                Size      Used Avail Use% Mounted on
/dev/mapper/vg_linuxl1st-lv_root
18G    4.9G    12G    30% /
tmpfs                    144M        0   144M    0% /dev/shm
/dev/sda1                 485M     33M   427M    8% /boot

192.168.37.142 | success | rc=0 >>
Filesystem                Size      Used Avail Use% Mounted on
/dev/mapper/vg_linuxl1st-lv_root
18G    5.4G    12G    33% /
tmpfs                    144M        0   144M    0% /dev/shm
/dev/sda1                 485M     33M   427M    8% /boot

192.168.37.156 | success | rc=0 >>
Filesystem                Size      Used Avail Use% Mounted on
/dev/mapper/vg_linuxl1st-lv_root
8.4G    6.4G    1.6G    81% /
tmpfs                    140M        0   140M    0% /dev/shm
/dev/sda1                 485M     35M   426M    8% /boot
/dev/sdb5                  20G     3.0G    16G   16% /data2
```

执行结果诠释：

以 192.168.37.130 的返回为例，success 表示命令执行成功，rc=0 表示 ResultCode=0，即命令返回结果，返回码为 0，表示命令执行成功，>> 后面跟的内容相当于在主机本地执行 df -lh 后的结果返回。

情景 2：批量查看远程主机内存使用情况（shell 模块）。

执行命令：

```
ansible apps -m shell -a "free -m"
```


返回结果如下：

```
192.168.37.142 | success | rc=0 >>
      total      used      free      shared    buffers     cached
Mem:      286      282         4         0         34        119
-/+ buffers/cache:      128        158
Swap:      2015        668       1347

192.168.37.130 | success | rc=0 >>
      total      used      free      shared    buffers     cached
Mem:      244      188         56         0         30        101
-/+ buffers/cache:         56        187
Swap:      1023         0       1023

192.168.37.155 | success | rc=0 >>
      total      used      free      shared    buffers     cached
Mem:      286      217         69         0         84         63
-/+ buffers/cache:         68        218
Swap:      2015         0       2015

192.168.37.156 | success | rc=0 >>
      total      used      free      shared    buffers     cached
Mem:      279      251         28         0         29         33
-/+ buffers/cache:      188         91
Swap:      1023        22       1001
```

执行结果诠释：

以 192.168.37.142 的返回为例，success 表示命令执行成功，rc=0 表示 ResultCode=0，即命令返回结果，返回码为 0，表示命令执行成功，>> 后面跟的内容相当于在主机本地执行 free-m 后的结果返回。

通过上面两个场景的示例相信大家大家对 Ad-Hoc 的用法有一定的了解，接下来的章节我们进一步学习 Ansible 的并发特性。

3.2.3 通过 Ad-Hoc 研究 Ansible 的并发特性

如 3.2.1 节所讲，Ansible 和 Ansible-playbook 默认会 fork 5 个线程并发执行命令，但在实际工作中，如果主机数量众多，Ansible 并发 5 个线程是远不能满足企业所需的，所以本节介绍 Ansible 的并发特性。我们通过如下测试来更深入地了解 Ansible 的并发工作模式。

场景如下：我们定义 [apps] 组，多次执行同样的 Ad-Hoc 命令来查看其返回的结果。

以下是执行步骤。

步骤 1：定义 [apps] 组，编辑 /etc/ansible/hosts 的配置。

执行命令 vi /etc/ansible/hosts，键入 i 进入 vi 编辑模式，跳转到文件最末尾，添加如下

```
配置：
[apps]
192.168.37.130
```

```
192.168.37.155 | success >> {
192.168.37.142 | success >> {
192.168.37.156 | success >> {
```

步骤 2: 多次执行 Ansible 命令, 执行命令如下:

```
ansible apps -m ping -f 3
```

步骤 3: 对比返回结果, 如表 3-1 所示。

表 3-1 返回结果对比

第 1 次返回结果	第 2 次返回结果
192.168.37.130 success >> { "changed": false, "ping": "pong" }	192.168.37.130 success >> { "changed": false, "ping": "pong" }
192.168.37.142 success >> { "changed": false, "ping": "pong" }	192.168.37.155 success >> { "changed": false, "ping": "pong" }
192.168.37.155 success >> { "changed": false, "ping": "pong" }	192.168.37.142 success >> { "changed": false, "ping": "pong" }
192.168.37.156 success >> { "changed": false, "ping": "pong" }	192.168.37.156 success >> { "changed": false, "ping": "pong" }

返回结果分析如下:

- 1) 同样的命令多次执行, 但每次的输出结果都不一定一样。
- 2) 输出结果不是按照 /etc/ansible/hosts 中 [apps] 定义的主机顺序输出。
- 3) 结果输出基本上遵循每次输出 3 条记录 (线程池始终保持 3 个线程, 所以这里如果每次输出小于等于 3 都是正常的)。

通过上面的实验我们对 Ansible 的并发性有了概念性的了解。回到前面的问题, 企业实际应用中, 如主机数量很多, 我们需调大线程数, 该如何操作呢? 这里 Ansible 为我们提供了便捷的选项, -f 指定线程数, 如 -f 1 表示并发启动一个线程, -f 10 则表示同时启动 10 个线程并发执行命令。其实查看源码可知, Ansible 使用 multiprocessing 管理多线程。

提示 单台主机的性能始终有限，大家根据自己机器实际的硬件配置做调整，建议并发数配置的 CPU 核数偶数倍就好。如 4Cores 8GB 的服务器，建议最多并发 20 个线程。关于 Ansible 的性能，后面章节会为大家介绍 Ansible 的加速模式。

3.2.4 通过 Ad-Hoc 研究 Ansible 的模块使用

前面的章节为大家详细介绍了 Ad-Hoc 的命令集使用方法及其并发特性等，那么，Ansible 究竟有多少现成功能可供大家使用呢？本节来为大家介绍 Ad-Hoc 的模块使用。

截至本篇编写时（2016-2-11），官方呈现的所有可用模块为 468 个（2016-8-19 所有可用模块为 622 个，短短 6 个月增加了 154 个，可见 Ansible 的发展速度），所有模块官方也做了详尽的功能分类。明细可参考官方链接^①。另外，Ansible 也提供了类似于 man 功能的 help 说明工具 `ansible-doc`，直接按回车键或输入 `-h` 显示功能用法。它和 Linux 系统下的 `man` 命令一样重要，正式学习 Ansible 模块使用前，有必要先了解 `ansible-doc` 用法。

命令用法：

```
ansible-doc [options] [module...]
```

可用选项如下。

- ❑ `--version`：显示工具版本号。
- ❑ `-h, --help`：显示该 help 说明。
- ❑ `-M MODULE_PATH, --module-path=MODULE_PATH`：指定 Ansible 模块的默认加载目录。
- ❑ `-l, --list`：列出所有可用模块。
- ❑ `-s, --snippet`：只显示 playbook 说明的代码段。
- ❑ `-v`：等同于一 `version`，显示工具版本号。

下面我们看些简单的示例。

情景 1：显示所有可用模块。

执行命令：

```
ansible-doc -l
```

返回结果如下：

<code>a10_server</code>	Manage A10 Networks AX/SoftAX/Thunder/vThunder devices
<code>a10_service_group</code>	Manage A10 Networks AX/SoftAX/Thunder/vThunder devices
<code>a10_virtual_server</code>	Manage A10 Networks AX/SoftAX/Thunder/vThunder devices
<code>acl</code>	Sets and retrieves file ACL information.
<code>add_host</code>	add a host (and alternatively a group) to the ansible-
<code>playboo...</code>	
<code>airbrake_deployment</code>	Notify airbrake about app deployments

① 模块链接：http://docs.ansible.com/ansible/modules_by_category.html。

alternatives	Manages alternative programs for common commands
apache2_module	enables/disables a module of the Apache2 webserver
apt	Manages apt-packages
apt_key	Add or remove an apt key
apt_repository	Add and remove APT repositories
apt_rpm	apt rpm package manager
arista_interface	Manage physical Ethernet interfaces
arista_l2interface	Manage layer 2 interfaces
arista_lag	Manage port channel (lag) interfaces
arista_vlan	Manage VLAN resources
assemble	Assembles a configuration file from fragments
...	

情景 2: 以 yum 模块为例, 我们希望获取 yum 模块的 HELP 说明。

执行命令:

```
ansible-doc yum
```

返回结果如下:

```
> YUM
  Installs, upgrade, removes, and lists packages and groups with the
  'yum' package manager.
```

Options (= is mandatory):

```
- conf_file
    The remote yum configuration file to use for the transaction.
    [Default: None]
...
```

其他模块 HELP 说明以此类推即可。下面通过 Ansible 内置模块来完成一些具体工作。

【示例 1】 安装 redhat-lsb 并查看服务器系统版本号。

步骤 1: 安装 redhat-lsb。

执行命令:

```
ansible apps -m yum -a 'name=redhat-lsb state=present'
```

返回结果如下:

```
192.168.37.142 | success >> {
  "changed": false,
  "msg": "",
  "rc": 0,
  "results": [
    "redhat-lsb-4.0-7.el6.centos.i686 providing redhat-lsb is already installed"
  ]
}
...
```


其中:

"changed": 主机是否有变更, true 为有; false 为没有(第1次运行或事先没有安装, 返回值一般是 true, 否则为 false)。

"msg": 安装过程信息。

"rc": 0, resultcode: 结果返回码, 非 0 返回码往往是红色并且错误的返回, 非常明显。

步骤 2: 查看系统版本号。

执行命令:

```
ansible apps -m command -a 'lsb_release -a'
```

返回结果如下:

```
192.168.37.155 | success | rc=0 >>
Version:          :base-4.0-ia32:base-4.0-noarch:core-4.0-ia32:core-4.0-noarch:graph
LSB
ics-4.0-ia32:graphics-4.0-noarch:printing-4.0-ia32:printing-4.0-noarch
Distributor ID: CentOS
Description:      CentOS release 6.5 (Final)
Release:          6.5
Codename:         Final
...
```

部分执行结果诠释:

192.168.37.155: 表示命令执行的对象。

success: 表示命令执行的返回状态为成功状态。

rc=0: 表示命令执行的状态码为 0。

>>: 该符号后返回的所有内容为执行 lsb_release -a 命令返回的信息。

LSB Version: 表示该系统的内核版本信息。

Distributor ID: 表示发行厂商。

Description: 表示版本简要信息。

Release: 表示该系统的发行版本号。

Codename: 表示发行版本代号。

【示例 2】为所有服务器安装 ntp 服务, 并设置为开机启动。

步骤 1: 安装 ntp 服务。

执行命令:

```
ansible apps -s -m yum -a "name=ntp state=present"
```

步骤 2: 启动 ntp 服务, 并设置为开机启动。

执行命令:

```
ansible apps -m service -a "name=ntpd state=started enabled=yes"
```

返回的结果不再一一为大家列举出来，相信上面那么多的示例，大家对如何判断结果是否正正确能够理解了。



如 Linux 所有命令的用法一样，我们只能记住日常工作中最常用到的那些，另外那些不常用的命令用法我们只需要知道如何快速获取它们的用法即可。Linux 系统为我们提供了 `man` 工具来快速获取所需。`ansible-doc` 则等同于 `man` 的功能和作用，这样解释相信大家会更容易理解其重要性。

3.3 Ad-Hoc 组管理和特定主机变更

3.2 节为大家介绍了 Ansible 模块列表及 HELP 说明获取方式。日常运维工作中，我们往往会将负责相同场景应用的主机划分为一个组，以方便统一管理。Ansible 也提供了简洁但强大的组管理功能。同时，我们也可能遇到只针对这组主机中一台或某些主机做变更的场景，针对这些复杂多变的企业场景，本节我们将深入了解 Ad-Hoc 组管理和特定主机变更，进一步了解 Ansible 如何应对复杂多变的企业环境。

3.3.1 Ad-Hoc 组定义

Ad-Hoc 的组功能定义在 Inventory 文件中，默认路径是 `/etc/ansible/hosts`，书写格式遵循 INI 风格，中括号中的字符为组名。可以将同一个主机同时归并到多个不同的组中；此外，若目标主机使用了非默认的 SSH 端口，还可以在主机名称之后使用冒号加端口号来标明。下面通过实际案例来了解 Inventory 文件的书写规则。

如下为 Inventory 文件示例，包括了组定义及冒号加端口号功能的使用。

```
ntp.magedu.com

[webservers]
www1.magedu.com:2222
www2.magedu.com

[dbservers]
db1.magedu.com
db2.magedu.com
db3.magedu.com
```

如果远程主机名称遵循相似的命名模式，还可以使用列表的方式标识各主机，如下案例为大家展示该写法。

```
[webservers]
www[01:50].magedu.com
```

```
[databases]
db-[a:f].magedu.com
```

本次架构规划了前端 Proxy、Web Servers 和后面 DB 一套完整应用，以方便大家实际参考，拓扑规划请参考图 3-5。图中共定义了 Proxy、App、NoSQL 和 DB 这 4 个组。组配置请编辑 /etc/ansible/hosts 添加如下配置：

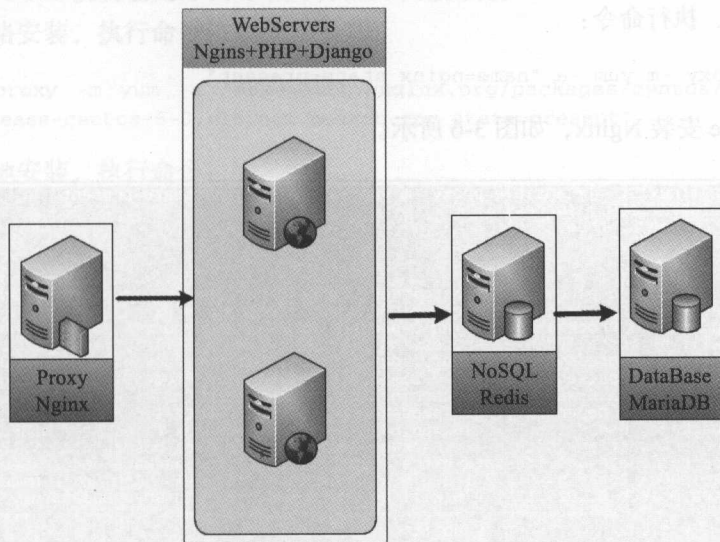


图 3-5 架构拓扑规划

```
[proxy]
192.168.37.159
[app]
192.168.37.130
192.168.37.160
[nosql]
192.168.37.142
[db]
192.168.37.142
```

应用分布如下。

- [proxy] 组：Nginx；
- [app] 组：Nginx+PHP+Django；
- [nosql] 组：Redis；
- [db] 组：Mariadb。

如图 3-5 所示的架构拓扑是简化后的互联网 Web 服务架构，用户请求通过 Proxy 转发至后端 WebServers 响应，通过 NoSQL 服务缓冲后，最终将请求传送到 DB。我们本章的实战内容就是通过 Ansible 来搭建这样一套 Web 服务架构。

3.3.2 Ad-Hoc 配置管理：配置 Proxy 与 Web Servers 实践

本节通过配置 Proxy & Web Servers 学习 Ad-Hoc 的组管理方式，前端 Proxy 只用到 Nginx 七层代理功能，将请求转发至后端 Web Servers，Web Servers 同时部署 Nginx、PHP 和 Django 应用，我们按 Proxy、WebServers 依次顺序部署应用。

(1) Ad-Hoc 配置管理 Proxy (即 Nginx)

安装 Nginx，执行命令：

```
ansible proxy -m yum -a "name=nginx state=present"
```

利用 Ansible 安装 Nginx，如图 3-6 所示。

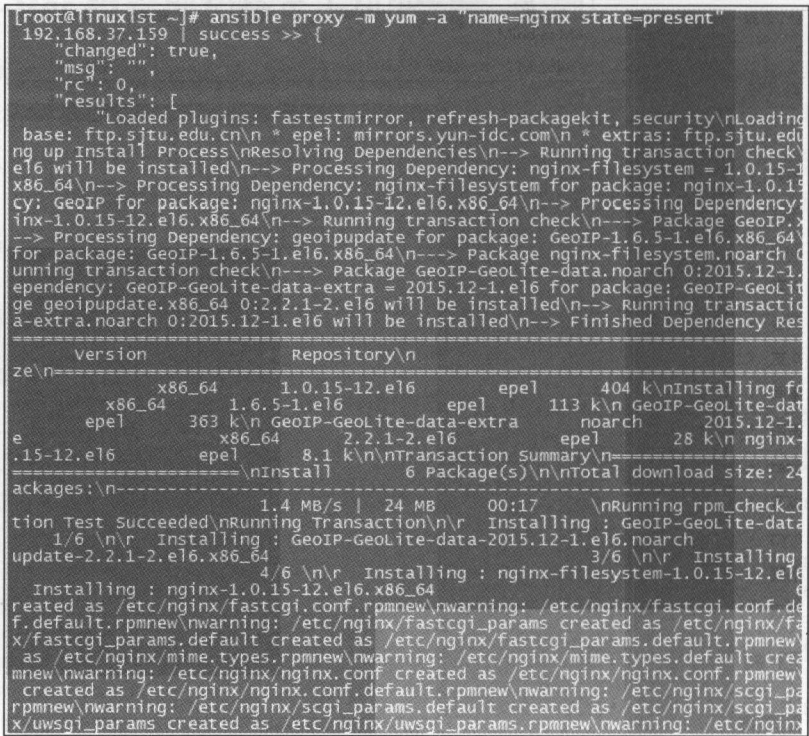


图 3-6 Ansible 安装 Nginx

部分执行结果诠释：

- "changed": true, // 表示本次命令对执行的目标有变更，如再执行一次则为 false，表示执行的目标没有变更，这里的 false 和 true 不代表该命令执行成功或失败，只是表示执行目标是否被变更
- "rc": 0, // resultcode 的简写，表示命令的执行结果状态返回，非 0 均为异常，命令执行失败
- "results": [// 执行结果信息返回

如果我们要检查 Nginx 正常安装，可以执行命令：ansible proxy -m command -a "nginx

-v", 如果正常则返回如下内容:

```
192.168.37.159 | success | rc=0 >>
nginx version: nginx/1.0.15
```

Ansible 的 YUM 模块同样支持指定某版本安装, 其 `name` 参数指定具体版本地址 (网络或本地均可)。YUM 模块也支持从网络安装或从本地安装。

如果从网络安装, 执行命令:

```
ansible proxy -m yum -a "name=http://nginx.org/packages/centos/6/noarch/RPMS/
nginx-release-centos-6-0.el6ngx.noarch.rpm state=present"
```

如果从本地安装, 执行命令:

```
ansible proxy -m yum -a "name=/usr/local/src/nginx-release-centos-6-0.el6ngx.
noarch.rpm state=present"
```

如指定网络安装, 安装期间请保障网络畅通, 并可以访问 Internet, 具体安装时长视网络宽带质量而定, 如 100M 带宽, 该项安装进行了约 2min。如图 3-6 为 Nginx 安装的部分结果反馈, Ansible 在对执行结果返回提示做得确实不尽人意, 虽为 JSON 格式输出, 但换行和颜色标识功能仍有待改善。

(2) Ad-Hoc 配置管理 Web Servers

Web Servers 需同时部署 Nginx、PHP 和 Django, 其中 Nginx、PHP 依然通过 YUM 模块实现, Django 推荐使用 PIP 或 `easy_install` 方式。

1) Nginx、PHP 安装命令如下:

```
ansible app -m yum -a "name=nginx state=present"
ansible app -m yum -a "name=php state=present"
```

2) Django 安装命令如下:

步骤 1: 安装 MySQL-python 和 python-setuptools 依赖包。

```
ansible app -m yum -a "name=MySQL-python state=present"
ansible app -m yum -a "name=python-setuptools state=present"
```

步骤 2: 安装 Django。

```
ansible app -m pip -a "name=django state=present"
```

步骤 3: 检查 Django 安装是否正常, 执行命令如下:

```
ansible app -m command -a "python -c 'import django; print django.get_
version()'"
```

其中 Django 依赖 Python 2.7+ 版本, 如执行报错如下, 请检查 Python 版本。

```
Traceback (most recent call last):
```

```

File "<string>", line 1, in <module>
File "/usr/lib/python2.6/site-packages/django/__init__.py", line 1, in <module>
from django.utils.version import get_version
File "/usr/lib/python2.6/site-packages/django/utils/version.py", line 7, in <module>
from django.utils.lru_cache import lru_cache
File "/usr/lib/python2.6/site-packages/django/utils/lru_cache.py", line 28
fasttypes = {int, str, frozenset, type(None)},
              ^
SyntaxError: invalid syntax

```



提示 PIP 和 easy_install 模块支持 virtualenv 虚拟多环境配置，该功能极为强大，是 Python 应用者的必备利器。如上我们使用 PIP 模块安装 Django，众所周知 easy_install 也是 Python 安装软件包常用的工具之一，Ansible 同样支持 easy_install，命令类同 # ansible app -m easy_install -a "name=django" 即可。两者相比较而言，PIP 的功能较 easy_install 更为强大，且支持卸载、指定版本号等。经过笔者亲测，建议使用 PIP 模块安装 Python 系列软件包，除功能强大外，我们发现 PIP 模块的稳定性和速度也远胜 easy_install 模块 (CentOS 6.5 Final)。

截至目前，Proxy & Web Servers 部署完毕，通过数条语句即可完成。本示例中的服务器只有 4 台，如果是 40、400、4000 台，其带来的便利和节省的时间成本是不可估量的，更为重要的是，这在很大程度上，可以保障操作的正确性。

其实上面的这些变更通过 Ansible-playbook 进行是更好的选择，在后面 playbook 章节的讲解与应用过程中大家会逐步有所体会。当然通过 shell 单独登录到对应的服务器也能实现我们想要完成的工作，但 Ansible 为我们节省了海量的时间成本，这也是我们学习 Ansible 的意义所在。

3.3.3 Ad-Hoc 配置后端：配置 NoSQL 与 Database Servers 实践

3.3.2 节我们使用 Ansible 配置了 Proxy 和 Web Servers，接下来我们使用类似的方法配置 NoSQL 和 DB 服务。我们使用 Redis 配置 NoSQL，我们使用 MariaDB 配置 DB，之所以没有使用 MySQL 是因为自从 MySQL 被甲骨文公司收购后，虽埃里森宣称 MySQL 依然免费，但随着时间的推移，MySQL 原班核心人马相继离开甲骨文，并创立 MariaDB 及 MySQL 社区，开发者对现有 Oracle 举动感到不满，行业内不少企业已在考虑使用 MariaDB，或计划替换现有 MySQL。

具体操作步骤如下。

Redis 安装命令：ansibledb -m yum -a "name=redis state=present"。

Redis 安装检查：ansibledb -m command -a "redis-cli --version"。

MariaDB 安装步骤如下。

步骤 1: 添加 yum 源, vim 编辑 /etc/yum.repos.d/mariadb.repo 添加内容如下。

```
# MariaDB 10.1 CentOS repository list - created 2016-02-13 04:31 UTC
# http://mariadb.org/mariadb/repositories/
[mariadb]
name = MariaDB
baseurl = http://yum.mariadb.org/10.1/centos6-x86
gpgkey=https://yum.mariadb.org/RPM-GPG-KEY-MariaDB
gpgcheck=1
```

步骤 2: 安装 MariaDB-server,

```
ansibledb -m yum -a "name=MariaDB-server state=present"
```

步骤 3: 安装 MariaDB-client,

```
ansibledb -m yum -a "name=MariaDB-client state=present"
```

步骤 4: 开启防火墙 3306 访问权限。

```
ansible db -m command -a "iptables -A INPUT -s 192.168.37.0/24 -p tcp -m tcp --dport 3306 -j ACCEPT"
```

截至目前,如图 3-5 所示的应用成功部署完毕。其实我们搭建一套主流 Web 应用框架,期间无需登录远程主机,通过 Ansible 的结果返回即可判断所有操作是否正确。

3.3.4 Ad-Hoc 特定主机变更

前两节我们通过搭建了一套主流 Web 应用框架熟悉了 Ansible 的组管理,接下来为大家介绍如何针对特定服务器做变更,该情景在日常运维工作中很常见。

Ansible 有多种方式实现针对特定主机做变更。

1) --limit: 通过 --limit 参数限定主机做变更。

情景: 在 App 组中启动 192.168.37.15 的 NTP 服务。

命令用法:

```
ansible app -m command -a "service ntpd status" --limit "192.168.37.158"
```

2) 指定 IP: 通过指定具体 IP 限定主机做变更。

情景: 启动 192.168.37.158 的 NTP 服务。

执行命令:

```
ansible 192.168.37.158 -m command -a "service ntpd status"
```

3) 用 ":" 作分隔符,指定多台机器做变更。

情景: 启动 192.168.37.158 和 192.168.37.161 的 NTP 服务。

执行命令:


```
ansible "192.168.37.158:192.168.37.161" -m command -a "service ntpd status"
```

4) 通过 “*” 泛匹配，更灵活地针对多台主机做变更。

情景：启动 192.168.37.* 所有主机的 NTP 服务。

执行命令：

```
ansible 192.168.37.* -m command -a "service ntpd status"
```

 **提示** --limit 在日常工作中经常用到，Ansible-playbook 也支持该参数。“:” 分隔指定多主机也是不错的办法，但记得要用引号将所有地址引起来。

到目前为止 Ad-Hoc 组管理及特定主机变更我们已经掌握，这对灵活管理海量服务器有很大帮助，关于组及指定主机管理介绍到此结束。接下来为大家介绍的是 Ad-Hoc 基于用户的管理。

3.4 Ad-Hoc 用户与组管理

用户权限管理是运维日常最重要的管理工作之一，如生产环境禁用开发和测试人员登录变更，但测试环境的用户权限仍需耗费精力维护，这项工作大公司也存在（将测试环境交给测试或开发管理并不是每个公司都能做到的，但未来是趋势）。所以掌握 Ad-Hoc 用户与组管理很有用，如笔者现在的公司每次大版本更新后都会大量修改所有服务器密码。每次需要修改数十台服务器环境密码，若手动单台登录修改可是一项不小的工作，并且手动方式难免会出错误。本节为大家介绍 Ad-Hoc 用户与组管理。

Ansible 系统用户模块有如下两个：

- ❑ Linux 系统用户管理：user。
- ❑ Windows 系统用户管理：win_user。

3.4.1 Linux 用户管理

User 模块功能诸多，各功能作用几乎完全覆盖平时工作常规及非常规场景。模块所有属性如表 3-2 所示。

表 3-2 user 模块属性

参 数	必填项	默认值	选 项	注 释
append	no	no	yes no	Yes: 增量添加 group no: 全量变更 group, 只设置 groups 指定的 group 组
comment	no			可选设置用户账户的描述（又名 GECOS）
createhome	no	yes	yes no	默认 yes, 当创建用户时期或家目录不存在时为用户创建 HOME 目录
expires (1.9 版本增加)	no	无		1.9 版本的新增功能，用户过期时间，不支持的平台该参数将被忽略，现在支持 Linux 和 FreeBSD

(续)

参 数	必填项	默认值	选 项	注 释
force	no	no	yes no	强制, 当和 state=absent 结合使用时, 效果等同于 userdel --force
generate_ssh_key	no	no	yes no	是否生成 SSH key, 不会覆盖已有的 SSH key
group	no			(可选) 设置用户属组
groups	no			设置用户附加群组, 使用逗号分隔多个群组, 如果参数为空 (即 'groups='), 则删除用户所有附加组 (属组不受影响)
home	no			(可选) 设置用户家目录
login_class	no			(可选) 设置 FreeBSD、OpenBSD、NetBSD 系统的用户登录 class
move_home	no	no	yes no	如设置为 yes, 结合使用 home=, 临时迁移用户家目录到特定目录
name	yes			用户名
non_unique	no	no	yes no	(可选) 和 -u 结合使用, 允许改变用户 ID 为非唯一值
password	no			(可选) 设置用户密码为该项指定的密码 (加密后的密码), 详细请参考 http://docs.ansible.com/ansible/faq.html#how-do-i-generate-crypted-passwords-for-the-user-module 需要注意的是, 在 Darwin 系统, 该选项必须是明文, 请注意安全问题
remove	no	no	yes no	结合 state=absent 使用相当于 userdel --remove
seuser (2.1 版本增加)	no			(可选) 设置 seuser 类型启用 SELinux
shell	no			(可选) 设置用户 shell
skeleton	no			(可选) 设置用户的 skel 目录, 需和 createhome 参数结合使用
ssh_key_bits	no	2048		(可选) 指定生成的 SSH key 加密位数
ssh_key_comment	no	ansible-generated on \$HOSTNAME		(可选) 定义 SSH key 注释
ssh_key_file	no	.ssh/id_rsa		(可选) 指定 SSH key 文件名, 如果该文件名是相对路径, 则默认路径为用户家目录
ssh_key_passphrase	no			设置 SSH key 密码, 如果没有提供密码, 则默认没有加密
ssh_key_type	no	rsa		(可选) 指定 SSH key 类型, 具体可用的 SSH key 类型取决于目标主机
state	no	present	present absent	Present: 新建 (使存在) 用户 absent: 删除用户
system	no	no	yes no	当创建新账户时, 该选项为 yes, 为用户设置系统账户, 该设置对已经存在的用户无效
uid	no			(可选) 设置用户 UID
update_password (1.3 版本增加)	no	always	always on_create	always: 只有当密码不相同时才会更新密码 on_create: 只为新用户设置密码

日常工作所需功能几乎均囊括在内，接下来为大家介绍用户相关的五大场景应用，以供参考。

场景 1：新增用户。

需求描述：新增用户 dba，使用 BASH Shell，附加组为 admins, dbagroup，家目录为 /home/dba/。

该场景中我们可以掌握如下技能点。

1) groups 设定：groups= 用户组 1，用户组 2……

2) 增量添加属组：append=yes

3) 表明属组状态为新建：state=present

执行命令：

```
ansibledb -m user -a "name=dba shell=/bin/bash groups=admins,dbagroup
append=yes home=/home/dba/ state=present"
```

返回结果如下：

```
192.168.37.142 | success >> {
    "changed": true,
    "comment": "",
    "createhome": true,
    "group": 503,
    "groups": "admins,dbagroup",
    "home": "/home/dba/",
    "name": "dba",
    "shell": "/bin/bash",
    "state": "present",
    "system": false,
    "uid": 501
}
```

返回结果信息非常简洁明了，这里不再一一做解释

场景 2：修改用户属组。

需求描述：修改 DBA 附件组为 dbagroups（即删除 admins 组权限）。

该场景中我们可以掌握如下技能点。

全量变更属组信息：append=no

执行命令：

```
ansibledb -m user -a "name=dba groups=dbagroup append=no"
```

返回结果如下：


```
192.168.37.142 | success >> {
    "append": false,
    "changed": true,
    "comment": "",

```

```

"group": 503,
"groups": "dbagroup",
"home": "/home/dba/",
"move_home": false,
"name": "dba",
"shell": "/bin/bash",
"state": "present",
"uid": 501
}

```

 **提示** 删除 admins 组权限的命令中 `append` 值为 `no`。另外，细心的朋友会发现，新增用户时，Ansible 默认为用户添加用户组（primary group）。

场景 3：修改用户属性。

需求描述：设置 dba 用户的过期时间为 2016/6/1 18:00:00 (UNIXTIME: 1464775200)。

该场景中我们可以掌握如下技能点。

- 1) 设置用户登录过期时间：`expire=1464775200`
- 2) UNIX 时间转换：2016/6/1 18:00:00 需转换为 UNIXTIME 格式（不做介绍，请自行

百度）

执行命令：

```
ansibledb -m user -a "name=dba expires=1464775200"
```

这样，我们已经完成了 DBA 用户的过期时间设置。

场景 4：删除用户。

需求描述：删除用户 DBA，并删除其家目录和邮件列表。

该场景中我们可以掌握如下技能点：

- 1) 表明属组状态为删除：`state=absent`
- 2) 设定 `remove=yes`；`remove=yes`

执行命令：

```
ansibledb -m user -a "name=dba state=absent remove=yes"
```

结果检查：到对应主机使用 ROOT 用户查看 `/etc/passwd` 是否存在 dba 用户，或执行命令 `id dba` 确认是否有结果返回。

场景 5：变更用户密码。

需求描述：设置系统用户 tom 的密码为 redhat123。

执行命令：

```
ansibledb -m user -a "name=tom shell=/bin/bash password=to46pW3G0ukvA update_password=always"
```

请注意，password 后的字符串 to46pW3GOukvA 并非真正的密码，而是经过加密后的密码。Ansible 变更用户密码方式与直接通过系统命令 `passwd` 修改密码有较大差别。Ansible 变更密码时所使用的密码是将明文密码加密后的密码（有些拗口）。官网上介绍了两种密码加密方式，笔者建议使用方式 2 `passlib`，因为 `mkpasswd` 方式因系统而异，功能差异较大。

方式 1：使用命令 `mkpasswd` 生成密码。

步骤 1：查找安装包名称。

执行命令：`yum whatprovides */mkpasswd`，结果如下。

```
epel/filelists_db | 8.0 MB
expect-5.44.1.15-5.el6_4.x86_64 : A program-script interaction and testing
utility
Repo : base
Matched from:
Filename : /usr/bin/mkpasswd
```

步骤 2：安装软件包。

centos 6.5 执行命令：`yum install expect`

Debian6 Ubuntu 12.04 执行命令：`sudo apt-get install whois`

步骤 3：使用 `mkpasswd` 生成密码。

执行命令：`mkpasswd --method=SHA-512`



提示 笔者也对安装的软件包（centos 安装 `expect`，debian&ubuntu 安装 `whois`）感觉诧异，一番 Google 查询也没有结果。因为这与本书内容关系不大所以没有深究，感兴趣的朋友自行研究后可告知笔者，以便后续补充给用户。

方式 2：使用 Python 的 `passlib`、`getpass` 库生成密码。

步骤 1：安装 `passlib`（Python 版本建议 2.7 以上）。

执行命令：

```
pip install passlib
```

步骤 2：生成密码。

Python 3.X 系列版本请使用如下命令（sha512 加密算法）。

```
python -c "from passlib.hash import sha512_crypt; import getpass; print (sha512_
crypt.encrypt(getpass.getpass()))"
```

Python 3.X 系列版本请使用如下命令（普通加密算法）。

```
python -c 'import crypt; print (crypt.crypt("redhat123", "dba"))'
```

Python 2.X 系列版本请使用如下命令（sha512 加密算法）。


```
python -c "from passlib.hash import sha512_crypt; import getpass; print sha512_crypt.encrypt(getpass.getpass())"
```


Python 2.X 系列版本请使用如下命令（普通加密算法）。

```
python -c 'import crypt; print (crypt.crypt("redhat123", "dba"))'
```

生成的密码如图 3-7 所示。

```
[root@linuxl1st ~]# python -c "from passlib.hash import sha512_crypt; import getpass; p
rint (sha512_crypt.encrypt(getpass.getpass()))"
Password:
$6$rounds=656000$NLhJPMjIGdlbLj3CsTjX0FL/QPvpTuDJPhnQKLIeaaBhin1NpYEd0/a5Yf0odWYYByrhH0
iLKDltSVP6BT0f9FrLGGzBMM5qN3.Hsdw.
```

图 3-7 生成的密码

-  **提示** 1) 同样密码多次加密结果不一样属正常情况，想深入了解的朋友可自行研究加密算法及原理。
- 2) Ad-Hoc 方式建议使用普通算法加密，sha512 加密后的密码包括诸多特殊元字符，传输至远程服务器会有密码被转义截断的问题。sha512 加密算法建议在 playbook 中使用。

Linux 系统下的用户与组管理涉及的各类场景本节均有介绍，作为运维日常最重要的工作之一，希望本节内容对相关人士会有帮助。

3.4.2 Windows 用户管理

如第 1 章所介绍，作为关注度最高的集中化管理工具，Ansible 同样支持 Windows 系统。但考虑 Windows 不开源的特殊性及服务器市场的占有率，使得 Ansible 与 Windows 的结合使用时总是会出问题，但其实类似问题其他工具也同样存在，这是 Windows 特性使然。本章我们只做简单演示，第 10 章我们还介绍 Windows 相关内容。

场景：新增用户 stanley，密码为 magedu@123，属组为 Administrators。

执行命令：

```
ansible windows -m win_user -a "name=stanley passwd=magedu@123
group=Administrators"
```

返回结果：

```
192.168.37.146 | success >> {
  "account_disabled": false,
  "account_locked": false,
  "changed": true,
  "description": "",
```

```

"fullname": "stanley",
"groups": [
    {
        "name": "Administrators",
        "path": "WinNT://WORKGROUP/LINUXLST/Administrators"
    }
],
"name": "stanley",
"password_expired": true,
"password_never_expires": false,
"path": "WinNT://WORKGROUP/LINUXLST/stanley",
"sid": "S-1-5-21-3965499365-1200628009-3594530176-1004",
"state": "present",
"user_cannot_change_password": false
}

```

部分返回结果诠释:

- ☐ account_disabled——禁用用户登录;
- ☐ account_locked——解锁用户;
- ☐ groups——用户所属组;
- ☐ name——用户名;
- ☐ password_expired——下次登录修改密码;
- ☐ user_cannot_change_password——用户是否可修改密码。

仅从操作上即可看出, Ansible 对 Windows 的用户管理也是基于 Linux 管理方式的沿用, 旨在简单易用。

3.4.3 应用层用户管理

前面两小节为大家介绍了 Ansible 基于 Linux 和 Windows 的系统管理。事实上, 除开源系统类 UNIX 系统和大家耳熟能详的 Windows 系统以外, Ansible 也支持商业系统或产品类应用, 系统如 AWS 的 IAM, MAC 的 OSX; 软件如 Apache CloudStack、Jabberd、OpenStack、MongoDB、MySQL、PostgreSQL、RabbitMQ、Vertica。本节我们以 MySQL 用户管理为例介绍。

情景: 新增 MySQL 用户 stanley, 设置登录密码为 magedu@bj, 对 zabbix.* 表有 ALL 权限。

执行命令:

```

ansible db -m mysql_user -a 'login_host=localhost login_password=magedu login_user=root name=stanley password=magedu@bj priv=zabbix.*:ALL state=present'

```

返回结果:

```

192.168.37.142 | success >> {
    "changed": true,

```

```

"user": "stanley"
}

```

登录验证步骤如下。

1) 在 db 服务器上测试登录。

```
mysql -ustanley -pmagedu@bj
```

2) 执行命令: `show grants for 'stanley'@'localhost';` 验证权限是否正确。

其实如上命令存在很大的安全隐患, 因为 MySQL 的登录信息完全暴露在命令台。

Ansible 建议的使用方式如下。

1) 在远程主机的 `~/.my.cnf` 文件中配置 root 的登录信息, 配置信息如下:

```

[client]
user=root
password=magedu

```

2) 命令行执行命令如下:

```
ansible db -m mysql_user -a 'name=stanley password=magedu@bj priv=zabbix.*:ALL
state=present'
```

此方式密码将不再暴露在控制台, 在一定程度上提高了服务安全性。

3.5 本章小结

Ansible Ad-Hoc 在运维日常工作中的作用举足轻重, 日常工作临时并发性操作均通过 Ad-Hoc 协助完成, 因此我们花了很多篇幅为大家介绍其使用及企业实践。同样重要的还有 Ansible 的 Playbook 功能, Ansible-playbook 可帮助我们完成更多、更复杂、更重要的功能。我们将在第 4 章、第 5 章、第 6 章深入学习 Ansible-playbook 的强大功能。

Playbook 快速入门

Ansible 使用 YAML 语法描述配置文件，YAML 语法以简洁明了、结构清晰著称。

Ansible 的任务配置文件被称为 Playbook，我们可以称之为“剧本”。每一个剧本（Playbook）中都包含一系列的任务，这每个任务在 Ansible 中又被称为“戏剧”（play）。一个剧本（Playbook）中包含多出戏剧（play），这很容易理解。

为了便于理解，再给大家举个现实中的例子。

NBA 球队教练手里都有一个叫战术板的東西，见图 4-1。每次暂停时，主教练都会在战术板上布置一系列战术（Playbook），球员在场上做出一系列的跑动和相互掩护动作来完成这个战术，这其中每一个跑位和掩护动作就可以被称为“play”。

在 Ansible 中，我们就充当编剧的角色，亲自编写剧本（一系列的服务端操作），让一出精彩的戏剧（play）巧妙配合，完成对服务器的一系列精确控制。

4.1 Playbook 语法简介

如本章一开始所讲，Playbook 采用 YAML 语法编写，YAML 不是一种标记语言。该语言在被开发时，它的意思其




图 4-1 篮球战术板

实是：Yet Another Markup Language（仍是一种标记语言）。结合 Ansible 中要用到的 YAML 语法点，我们对 YAML 语法简洁地总结如下。

4.1.1 多行缩进

数据结构可以用类似大纲的缩排方式呈现，结构通过缩进来表示，连续的项目通过减号“-”来表示，map 结构里面的 key/value 对用冒号“:”来分隔。格式如以下代码所示：

```
house:
family:
name: Doe
parents:
  - John
  - Jane
children:
  - Paul
  - Mark
  - Simone
address:
number: 34
street: Main Street
city: Nowheretown
zipcode: 12345
```

注意

- 1) 字符串一定要用双引号标识；
- 2) 在缩排中空白字符的数目并不重要，只要相同阶层的元素左侧对齐就可以了（不过不能使用 Tab 字符）；
- 3) 允许在文件中加入选择性的空行，以增加可读性；
- 4) 选择性的符号“...”可以用来表示档案结尾（在利用串流的通信中，这非常有用，可以在不关闭串流的情况下，发送结束信号）。

4.1.2 单行缩写

YAML 也有用来描述好几行相同结构的数据的缩写语法，数组用“[]”包括起来，hash 用“{}”来包括。因此，上面的这个 YAML 能够缩写成这样：

```
house:
  family: { name: Doe, parents: [John, Jane], children: [Paul, Mark, Simone] }
  address: { number: 34, street: Main Street, city: Nowheretown, zipcode: 12345 }
```

了解了普通的 YAML 格式文件，我们来看一下正式的 Playbook 内容是什么样的。Playbook 代码如下：

```

---
# 这个是你选择的主机
- hosts: webservers
# 这个是变量
vars:
    http_port: 80
    max_clients: 200
# 远端的执行权限
remote_user: root
tasks:
# 利用 YUM 模块来操作
- name: ensure apache is at the latest version
yum: pkg=httpd state=latest
- name: write the apache config file
template: src=/srv/httpd.j2 dest=/etc/httpd.conf
# 触发重启服务器
notify:restart apache
- name: ensure apache is running
service: name=httpd state=started
# 这里的 restart apache 和上面的触发是配对的。这就是 handlers 的作用。相当于 tag
handlers:
- name: restart apache
service: name=httpd state=restarted

```

总的来说，Playbook 语法具有如下一些特性。

- 1) 需要以 “---” (3 个减号) 开始，且需顶行首写。
- 2) 次行开始正常写 Playbook 的内容，但笔者建议写明该 Playbook 的功能。
- 3) 使用 # 号注释代码。
- 4) 缩进必须是统一的，不能将空格和 Tab 混用。
- 5) 缩进的级别必须是一致的，同样的缩进代表同样的级别，程序判别配置的级别是通过缩进结合换行来实现的。
- 6) YAML 文件内容和 Linux 系统大小写判断方式保持一致，是区别大小写的，k/v 的值均需大小写敏感。
- 7) k/v 的值可同行写也可换行写。同行使用 “:” 分隔，换行写需要以 “-” 分隔。
- 8) 一个完整的代码块功能需最少元素，需包括 name: task。
- 9) 一个 name 只能包括一个 task。

4.2 Playbook 案例分析

第 3 章中介绍的 Ad-Hoc 点对点、一次执行一个模块的操作方式已经使得 Ansible 成为非常强大的管理工具，但 Playbook 将会使 Ansible 的功能更为完善、Playbook 可以固化所有操作，减少人为失误，满足工程量较大的工具所具备的复杂度、可扩展度等要求，使得 Ansible 成为超一流的管理工具。

Shell 脚本与 Playbook 的转换

现在，越来越多的 DevOps 也开始将目光移向了 Ansible，因为 Ansible 可以轻松地将 Shell 脚本或简单的 Shell 命令转换为 Ansible play。

下面一段代码是一个安装 Apache 的 Shell 脚本，大家来感受一下。

```
# !/bin/bash
# 安装 Apache
yum install --quiet -y httpd httpd-devel
# 复制配置文件
cp /path/to/config/httpd.conf /etc/httpd/conf/httpd.conf
cp/path/to/httpd-vhosts.conf /etc/httpd/conf/httpd-vhosts.conf
# 启动 Apache，并设置开机启动
service httpd start
chkconfig httpd on
```

将上述 Shell 脚本转换为一个完整的 Playbook 后，内容如以下代码所示：

```
---
- hosts: all
  tasks:
    - name: "安装 Apache"
      command: yum install --quiet -y httpd httpd-devel
    - name: "复制配置文件"
      command: cp /tmp/httpd.conf /etc/httpd/conf/httpd.conf
      command: cp /tmp/httpd-vhosts.conf /etc/httpd/conf/httpd-vhosts.conf
    - name: "启动 Apache，并设置开机启动"
      command: service httpd start
      command: chkconfig httpd on
```

将以上内容放在一个名为 `playbook.yml` 的文件中，直接调用 `ansible-playbook` 命令，即可运行，运行结果和脚本运行结果一致。

```
# ansible-playbook ./playbook.yml
```

也就是说，只要有编写 Shell 脚本的基本能力，你就可以快速学会利用 Playbook 来发挥 Ansible 的强大威力。

在上述 Playbook 中，我们使用了 `command` 模块来运行了标准的 Shell 命令。我们还给了每一出 `play` 一个“`name`”，因此当我们运行 Playbook 时，每一个 `play` 都会有非常易读的信息输出。

上面的 Playbook 已经很好地实现了与 Shell 脚本相同的功能，但是 Ansible 还有很多其他内置模块，可以大幅度提升处理复杂配置的能力，如代码清单 4-1 所示。

代码清单 4-1 部署 Apache 服务

```
---
- hosts: all
  sudo: yes
```

```

tasks:
  - name: 安装 Apache
  yum: name={{ item }} state=present
    with_items:
      - httpd
      - httpd-devel
      - name: 复制配置文件
  copy:
    src: "{{ item.src }}"
    dest: "{{ item.dest }}"
    owner: root
    group: root
    mode: 0644
    with_items:
      - {
        src: "/tmp/httpd.conf",
        dest: "/etc/httpd/conf/httpd.conf" }
      - {
        src: "/tmp/httpd-vhosts.conf",
        dest: "/etc/httpd/conf/httpd-vhosts.conf"
      }
      - name: 检查 Apache 运行状态，并设置开机启动
  service: name=httpd state=started enabled=yes

```

本例中，我们使用了 Ansible 的 yum 模块、copy 模块以及 service 模块，来完成对 Apache 服务的安装、配置和运行状态维护等一系列操作。

现在我们已经对 Playbook 有了一个大致的了解，接下来，让我们详细解剖一下代码清单 4-1 中的 Playbook 都做了什么，以及它是如何工作的。

第 1 行，“---”，这个是 YAML 语法中注释的用法，就像 shell 脚本中的“#”号一样。

第 2 行，“- hosts: all”，告诉 Ansible 具体要在哪些主机上运行我的剧本（Playbook），在本例中是 all，即所有主机。

第 3 行，“sudo: yes”，告诉 Ansible 通过 sudo 来运行相应命令，这样所有命令将会以 root 身份执行。

第 4 行，“tasks:”，指定一系列将要运行的任务。

每一个任务（play）以“- name: 安装 Apache”开头。“- name:”字段并不是一个模块，不会执行任务实质性的操作，它只是给“task”一个易于识别的名称。即便把 name 字段对应的行完全删除，也不会有任何问题。

本例中我们使用 yum 模块来安装 Apache，替代了“yum -y install httpdhttpd-devel”。

在每一个 play 当中，都可以使用 with_items 来定义变量，并通过“{{ 变量名 }}”的形式来直接使用。我们使用 yum 模块的 state=present 选项来确保软件被安装，或者使用 state=absent 来确保软件被删除。

第二个任务（play）同样是“- name”字符开头。我们使用 copy 模块来将“src”定义的

源文件（必须是 Ansible 所在服务器上的本地文件）复制到“dest”定义的目的地址（此地址为远程主机上的地址）去。在传递文件的同时，还定义了文件的属主、属组和权限。在这个 play 中，我们用数组的形式给变量赋值，使用 {var1: value, var2: value} 的格式来赋值，变量的个数可以任意多，不同变量间以逗号分隔，使用 {{item.var1}} 的形式来调用变量，本例中为 {{item.src}}。

第三个任务（play）使用了同样的结构，调用了 service 模块，以保证服务的正常开启。

4.3 Playbook 与 Shell 脚本差异对比

当我们把 Shell 脚本转换为 Playbook 运行的时候，Ansible 会留下清晰的执行痕迹，明确告诉我们在每一台主机上的每一步都做了什么。

同时，Ansible 自带幂等判断机制也为运维省去不少伤脑费心的人脑逻辑判断运算。当我们重复执行一个 Playbook 时，当 Ansible 发现系统的现有状态与 Playbook 所定义的将要实现的状态一致时，Ansible 将自动跳过该操作。

我们再次执行 Playbook: temp.yml，当 Ansible 发现 Playbook 中的 play 都已完成时，它将直接返回 ok 状态码，速度非常之快。如果是 Shell 脚本，肯定会把所用操作再做一遍。

在正式运行 Playbook 之前，可以使用 --check 或 -C 选项来检测 Playbook 都会改变哪些内容，显示的结果跟真正执行时一模一样，但不会真的对被管理的服务器产生实际影响。

4.4 Ansible-playbook 实战小技巧

本节将介绍一些实战的小技巧，帮助我们在 Playbook 之外对 Ansible-playbook 命令的执行范围以及权限进行微调。

4.4.1 限定执行范围

当 Playbook 指定的一批主机中有个别主机需进行变更时，我们不需要去修改 Playbook 文件本身，而是通过一些选项就可以直接限定和查看 Ansible 命令的执行范围。

1. --limit

如果运行上面的例子，会发现所有被 Ansible 管理的主机都会被操作。

我们可以通过修改“- hosts:”字段来指定哪些主机将会应用 Playbook 的操作。

□ 指定一台主机：www.magedu.com

□ 指定多台主机：www.magedu.com, www.osstep.com

□ 指定一组主机：dbserver

当然，也可以直接通过 ansible-playbook 命令来指定主机：

```
# ansible-playbookplaybook.yml --limit webserver
```

这样一来（假设你的 inventory 文件中包含 webserver 组），即便 Playbook 中设定 “hosts: all”，但也仅对 webserver 组生效。

2. --list-hosts

如果想知道在执行 Playbook 时，哪些主机将会受影响，则使用 --list-hosts 选项。

```
# ansible-playbookplaybook.yml --list-hosts
```

运行结果：

```
playbook: playbook.yml
play # 1 (all): host count=1
webserver
```

如上两种方式针对需对批量操作的主机列表中的某一台主机做特定修改时非常有用。该技巧极大地增加了 Ansible 的使用灵活性。



注意 因为用于测试的被管理主机只有一台，所以 count 结果为 1。

4.4.2 用户与权限设置

(1) --remote-user

在 Playbook 中，如果在 hosts 字段下没有定义 users 关键字，那么 Ansible 将会使用你在 Inventory 文件中定义的用户，如里 Inventory 文件中也没定义用户，Ansible 将默认使用当前系统用户身份来通过 SSH 连接远程主机，在远程主机中运行 play 内容。

我们也可以直接在 ansible-playbook 中使用 --remote-user 选项来指定用户。

```
# ansible-playbookplaybook.yml --remote-user=tom
```

(2) --ask-sudo-pass

在某些情况下，我们需要传递 sudo 密码到远程主机，来保证 sudo 命令的正常运行。这时，可以使用 --ask-sudo-pass (-K) 选项来交互式的输入密码。

(3) --sudo

使用 --sudo 选项，可以强制所有 play 都使用 sudo 用户，同时使用 --sudo-user 选项指定 sudo 可以执行哪个用户的权限，如果不指定，则默认以 root 身份运行。

比如，当前用户 Tom 想以 Jerry 的身份运行 Playbook，命令如下：

```
$ansible-playbookplaybook.yml --sudo --sudo-user=jerry --ask-sudo-pass
```

执行过程中，会要求用户输入 Jerry 的密码。

4.4.3 Ansible-playbook: 其他选项技巧

Ansible-playbook 命令还有一些其他选项。

- ❑ `--inventory=PATH (-i PATH)`: 指定 inventory 文件，默认文件是 `/etc/ansible/hosts`。
- ❑ `--verbose (-v)`: 显示详细输出，也可以使用 `-vvvv` 显示精确到每分钟的输出。
- ❑ `--extra-vars=VARS (-e VARS)`: 定义在 Playbook 使用的变量，格式为: `"key=value, key=value"`。
- ❑ `--forks=NUM (-f NUM)`: 指定并发执行的任务数，默认为 5，根据服务器性能，调大这个值可提高 Ansible 执行效率。
- ❑ `--connection=TYPE (-c TYPE)`: 指定连接远程主机的方式，默认为 SSH，设为 `local` 时，则只在本地执行 Playbook，建议不做修改。
- ❑ `--check`: 检测模式，Playbook 中定义的所有任务将在每台远程主机上进行检测，但并不真正执行。

以上这些参数可满足大部分的工作需求。

4.5 实战一: Ansible 部署 Node.js 企业实践

我们在代码清单 4-1 中展示过一个简单搭建的 WEB 平台，这个例子用于做一些最基本和静态页面展示还是可以的，但是并不适用于正式的生产环境。在接下来的 3 节中，我们将会带来更多功能更为完善的实战项目，其中大部分都是在实际生产中被验证过的。

下面将要在我们的 CentOS6.x 服务器上配置 Node.js，启动一个简单的 Node.js 实例，其架构如图 4-2 所示。

项目首先创建一个尽可能简单的 playbook 文件，如代码清单 4-2 所示。

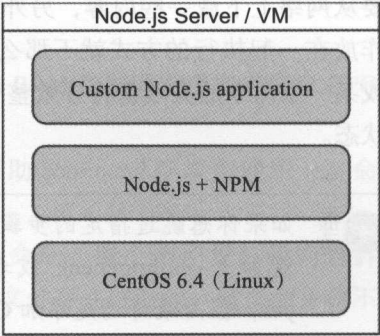


图 4-2 Node.js 服务架构

```
代码清单 4-2 Playbook 通用头部信息
---
- hosts: all
tasks:
```

接下来定义要进行部署的主机以及将要运行的任务。

4.5.1 添加第三方源

在准备部署一个服务器的时候，为了确保指定的软件包可用或者是最新的版本，管理员经常首先添加额外的源（YUM 源或 APT 源）。

下面的脚本，我们想要添加 EPEL 和 Remi 源，这些源中包含了最新版的 Node.js 的软件包，当然同时也可以用于为其他软件更新。我们使用如下脚本可以完成源的添加和 Node.js 软件的安装工作。

```
#!/bin/bash
# 导入 Remi GPG 密钥
wget http://rpms.famillecollet.com/RPM-GPG-KEY-remi \
-O /etc/pki/rpm-gpg/RPM-GPG-KEY-remi
rpm --import /etc/pki/rpm-gpg/RPM-GPG-KEY-remi

# 安装 Remi 源
rpm -Uvh --quiet \
http://rpms.famillecollet.com/enterprise/remi-release-6.rpm

# 安装 EPEL 源
yum install epel-release

# 安装 Node.js (npm + 和它的依赖关系)
yum --enablerepo=epel install npm
```

这个 Shell 脚本首先导入了 EPEL 和 Remi 的 GPG keys，然后安装这些到本地，最后安装 Node.js。这种使用 Shell 脚本的方法对于简单的部署是没有问题的，但是在本例中我们需要从网络上下载一些内容，另外还有一些操作是比较费时的，那么 Shell 脚本这种将很多工作放在一起执行的方式就不那么妥当了。因为在这个耗时很长的运行过程中，网络的中断或某一条命令的失败都将导致整个脚本的中断，而且是一部分运行成功，一部分运行失败的状态。



提示 如果你想跳过指定的步骤，可以跳过添加 GPG keys 的步骤，只需要在运行命令的时候加上 `--nogpgcheck`。或者在 Ansible 中，yum 模块中设置 `disable_gpg_check` 参数为 `yes`，但是最好还是添加 GPG keys。使用 GPG，你可以知道包的作者是谁，包有没有修改稿，除非你知道你正在做什么，否则最好不要禁止 GPG 检查。

利用 Ansible 可以让这一过程变得更具健壮性。下面我们介绍使用 Ansible 的案例。它和上面的 Shell 脚本有同样的功能，但是更容易理解，结构也更加清晰。我们同时还使用了 Ansible 的变量和其他的一些有用的特性。接着前面代码清单 4-2 中我们写好的 Playbook 开头，继续往下写。

```
tasks:
  - name: 导入 Remi GPG 密钥
    rpm_key: "key={{ item }} state=present"
    with_items:
      - "http://rpms.famillecollet.com/RPM-GPG-KEY-remi"
  - name: Install Remi repo.
```



```

command: "rpm -Uvh --force {{ item.href }} creates={{ item.create }}"
  with_items:
    - href: "http://rpms.famillecollet.com/enterprise/remi-release-6.rpm"
  creates: "/etc/yum.repos.d/remi.repo"

- name: 安装 Remi 源
yum: name=epel-release state=present

- name: 关闭防火墙
service: name=iptables state=stopped

- name: 安装 NodeJS 和 npm
yum: name=npm state=present enablerepo=epel

- name: 使用 Taobao 的 npm 源
command: >
npm config set registry https://registry.npm.taobao.org

- name: 关闭 npm 的 https
command: >
npm config set strict-ssl false

- name: 安装 Forever (用于启动 Node.js app)
npm: name=forever global=yes state=latest

```

我们看一下具体步骤。

第1个任务，rpm_key 是一个 Ansible 模块，用于从你的 RPM 数据库中添加或移除 GPG key。我们正在从 Remi 的源中导入一个 key。

第2个任务，因为 Ansible 没有 rpm 命令，因此我们借助 command 模块来使用 rpm 命令，这样我们可以做其他的两件事情。

1) 使用 create 参数告诉 Ansible 什么时候不运行这个命令。这个例子里，我们告诉 Ansible，这个命令成功执行后，将会创建那些文件。当这个文件存在的时候，这个命令将不会运行。

2) 使用 with_items 定义一个 URL 和用于 creates 检查的文件。

第3个任务，全用 yum 模块安装 EPEL 源。

第4个任务，因为这个服务器我们将用作测试，所以我们使用 service 模块禁止系统防火墙，防止它干涉我们测试。

第5个任务，使用 yum 模块来安装 Node.js 和 npm (Node 的包管理器)，我们使用 enablerepo 指定在 EPEL 源中搜索它，当然也可以使用 disablerepo 指定不使用哪个源 (repository)。

第6个任务，我们使用 Taobao 的 npm 源替换默认的国外源。

第7个任务，我们禁用了 https 检测，因为在有些环境中会报告一些 ssl 相关的异常，但是这里我们有理由相信 Taobao 源的可靠性。

第 8 个任务，因为我们现在已经安装了 npm，所以可以使用 Ansible 的 npm 模块安装 Node.js 的管理工具 forever 来运行我们的 Node.js app，设置 global 为 yes，指定模块的安装位置为 /usr/lib/node_modules，然后所有的用户都可以使用这些模块。

到此为止，我们已经有一个 Node.js app 服务器了，接下来让我们部署一个简单的 Node.js app，使用 80 端口来响应 HTTP 请求。

Node.js 应用部署。首先，通过创建一个新的名为 app 的文件夹，这个文件夹和我们上一步创建的 yml 文件处于相同的路径下面。然后在这个文件夹里面创建文件 app.js，并编辑其内容如下：

```
// 加载 express 模块
var express = require('express'),
    app = express.createServer();

// 响应 "/" 请求为 'Hello World'
app.get('/', function(req, res){
    res.send('Hello World! Yunzhonghe');
});

// 在 80 端口监听
app.listen(80);
console.log('Express server started successfully.')
```

不会 Node.js 语法？没关系！这个案例也可以用 Python、Perl、Java、PHP，或者其他编程语言来写。但是因为 Node 是非常简单的语言并且轻量级的系统，所以它是一个非常不错的用于测试你的服务器语言。

因为这个小 app 依赖于 Express（一个简单的 Node 的 HTTP 框架），我们还需要通过一个 package.json 文件告诉 NPM 关于它的依赖关系，这个文件与 app.js 处于相同的路径下面。

```
{
  "name": "examplennodeapp",
  "description": "Example Express Node.js app.",
  "author": "yunzhonghe",
  "dependencies": {
    "express": "3.x.x"
  },
  "engine": "node >= 0.10.6"
}
```

然后添加下面内容到你的 Playbook 里面，这段代码将复制整个 app 目录到目标服务器，然后使用 npm 下载安装所依赖的文件（这里为 express.），具体见代码清单 4-3。

代码清单 4-3 传输 app 目录并安装依赖文件


```
- name: 确保 Node.js app 的目录存在
  file: "path={{ node_apps_location }} state=directory"
```

```
- name: 复制 Node.js app 整个目录到目标主机
copy: "src=app dest={{ node_apps_location }}"
```

```
- name: 安装 package.json 文件中定义的依赖关系
npm: "path={{ node_apps_location }}/app"
```

首先我们使用 file 模块确保我们安装的 app 目录存在。{{node_apps_location}} 变量可以在 vars 部分定义，vars 部分位于 playbook 的顶部。当然也可以在 Inventory 文件中定义，也可以在运行 ansible-playbook 的时候定义。

我们使用 Ansible 的 copy 模块复制整个 app 目录到测试服务器，copy 模块可以自动区分单一的文件和包含文件的目录，然后在目录中递归，就像 rsync 或 scp。

 **提示** Ansible 的 copy 模块在单个文件或少量文件时候非常好用，但是如果复制大量的文件，嵌套几层的目录，copy 模块就不能胜任了。这种情形下，如果你想复制整个目录，最好使用 synchronize 模块，如果你想复制一个归档，然后展开它，最好使用 unarchive 模块。

最后，我们使用 npm 模块，这次除了 app 的路径之外没有额外的参数。这告诉 NPM 来解析 package.json 文件，然后确保所有的依赖关系都存在。

下面要做的就是启动这个 app。

4.5.2 运行 Node.js 进程

我们现在使用 forever 命令来启动这个 app。

```
- name: 获取正在运行的 Node.js app 列表
command: forever list
register: forever_list
changed_when: false
```

```
- name: 启动 Node.js app
command: "forever start {{ node_apps_location }}/app/app.js"
when: "forever_list.stdout.find('{{ node_apps_location }}/app/app.js') == -1"
```

在第 1 个任务中，我们做了两件新的事情。

1) register 创建了一个新的变量 forever_list，以便于下次任务的时候用作判断条件。register 用于保存命令的输出（包括标准输出和错误输出），然后赋给变量名。

2) changed_when 可以在任务运行完成后，明确告诉 Ansible 这个任务是否对主机造成了影响（比如改变了文件或安装了软件等）。在本次任务中，forever list 命令永远都不会导致服务器的改变，所以我们指定其值为 false。

第 2 个任务实际上使用 forever 启动了这个 app。我们也可以通过调用 node {{node_

`apps_location}}/app/app.js` 启动这个 app，不过这种方式更难控制。

`forever` 会一直跟踪它所管理的 Node app，然后我们使用 `forever` 的 `list` 选项打印正在运行的 app 列表。我们第一次运行这个 Playbook 时候，因为之前从未启动过 Node.js app，所以这个列表将会是空的。我们使用 `when` 语句来判断 app 的路径是否在 `forever list` 的输出信息中，如果不存在，则表明我们的 Node.js app 还未启动，于是触发任务启动之。

4.5.3 Node.js app 服务部署总结

到这个时候，我们已经完成了可以安装一个简单的可以通过 80 端口响应 HTTP 请求的 Node.js app。

我们可以通过如下命令来使我们编辑的 Playbook 在服务器上生效，同时使用 `--extra-vars` 选项来对代码清单 4-3 中的变量 `node_apps_location` 进行赋值。

```
ansible-playbook --extra-vars="node_apps_location=/usr/local/opt/node"
```

当这一切所有操作都完成之后，在浏览器中访问 Node.js 主机名查看效果，测试页面如图 4-3 所示。

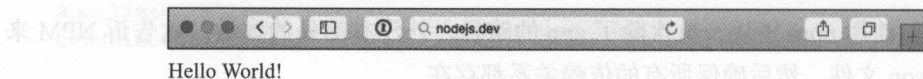


图 4-3 Node.js 测试页面

简单，但是有效，我们已经可以在一个少于 50 行的 YML 文件中配置一个 Node.js 应用服务器了。

4.6 实战二：Drupal 基于 LAMP 的自动化部署

Drupal 是使用 PHP 语言编写的开源内容管理框架 (CMF)，它由内容管理系统 (CMS) 和 PHP 开发框架 (Framework) 共同构成。它连续多年荣获全球最佳 CMS 大奖，是基于 PHP 语言最著名的 Web 应用程序。本节我们借助 Drupal 的自动化部署的实际案例，来介绍应用范围更广的 LAMP (Linux、Apache、MySQL、PHP) 的自动化实现和 Ansible 的更多实用技巧。

Ansible 对多种类型的 Linux 系统都有很好的支持。本节我们就以 Ubuntu 12.04 (版本 14.04 中的部署过程相同) 为主机系统，来实现 Ansible 自动化部署使用 Drupal 框架的 LAMP。系统架构如图 4-4 所示。

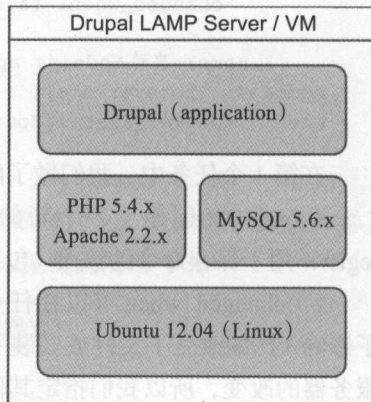


图 4-4 LAMP 架构图

4.6.1 定义变量并设置 Handlers

在 Playbook 中使用变量文件来定义变量，这样可以使 Playbook 看起来更加整洁，同时也能提高效率。我们首先使用下面的语法在 Playbook 文件 `playbook.yml` 头部添加对变量文件的引用。

```
---
- hosts: all
vars_files:
  - vars.yml
```

变量文件可以实现变量的集中管理，使得变量的管理更加方便、高效。现在我们的项目中还没有变量需要定义，在接下来的部署中，当有变更需要定义的时候，我们将直接编辑变量文件 `vars.yml` 来实现变量的定义。


Ansible 中，我们可以在 Playbook 中使用 `pre_task` 和 `post_task` 指定在主任务运行之前和之后要运行的任务。在本例中，需要确保在我们正式开始运行部署 LAMP 的任务之前，APT 缓存是被更新过的，也就是在任务开始前我们主机上的软件包都必须是最新的。这时，我们将得用 `apt` 模块来更新 APT 缓存，同时设置缓存有效期为 3600 秒。实现方法如下：

```
pre_tasks:
  - name: Update apt cache if needed.
    apt: update_cache=yes cache_valid_time=3600
```

解决了 APT 缓存的问题之后，我们将借助 `handlers`（将在第 5 章中详细介绍）来实现对 `apache2` 服务的启动管理。

```
handlers:
  - name: restart apache
    service: name=apache2 state=restarted
```

Handlers 是 Playbook 中一种特殊的任务类型，我们通过在任务末尾使用 `notify` 选项加 Handlers 的名称，来触发对应名称下 Handlers 中定义的任务。在本例中，我们将在 Apache 配置完成后，或 Apache 配置文件变动过后，使用 `notify: restart apache` 来调用 handler 重启 Apache 服务。

 提示 就像变量可以在独立文件中定义一样，Handlers 以及 Playbook 中的任务也可以在独立的文件中进行定义，然后被当前的 Playbook 进行引用（将会在第 6 章中详细介绍），以保证 Playbook 内容的条理性。在本例中，为了保持简单，我们还是将 Handlers 以及 Playbook 任务的定义放在了一个 Playbook 中。我们将会在今后的章节中探讨 Playbook 不同的组织方式。

默认情况下，当一个 Playbook 中的任务执行失败时，Ansible 会停止所有的任务，而且

也不会再触发任何本该被触发的 Handlers。在某些情况下，这种默认机制会造成不可预测的负面影响。如果我们要确认 Handlers 无论如何都要被正常触发的话，可以在使用 `ansible-playbook` 命令执行 Playbook 时，使用 `--force-handlers` 选项来强制要求 Handlers 可以被正常触发。

4.6.2 部署 LAMP 基础服务

部署一个基于 LAMP 的应用的第一步就是搭建 LAMP 服务本身。这是最简单也是最基础的一步。在开始部署之前，我们还需要根据具体要求做一个前期准备工作。我们需要安装 Apache、MySQL 及 PHP 到服务器上面，这之前需要解决一些依赖关系。我们需要安装 5.5 版本的 PHP 软件，但是这个版本的 PHP 包含在默认的 APT 源中，所以我们还要添加一个包含 PHP 5.5 版本的外部 APT 源。

tasks:

- name: " 安装用来管理 APT 源的工具 "
 - apt: name={{ item }} state=present
 - with_items:
 - python-apt
 - python-pycurl
- name: " 添加包含 5.5 版本 PHP 的 ondrej 源 "
 - apt_repository: repo='ppa:ondrej/php5' update_cache=yes
- name: " 安装 Apache、MySQL、PHP，以及依赖关系 "
 - apt: name={{ item }} state=present
 - with_items:
 - git
 - curl
 - sendmail
 - apache2
 - php5
 - php5-common
 - php5-mysql
 - php5-cli
 - php5-curl
 - php5-gd
 - php5-dev
 - php5-mcrypt
 - php-apc
 - php-pear
 - python-mysqldb
 - mysql-server
- name: " 关闭防火墙（因为本项目仅供本地开发使用） "
 - service: name=ufw state=stopped
- name: " 启动 Apache、MySQL 及 PHP "

```

service: "name={{ item }} state=started enabled=yes"
with_items:
  - apache2
  - mysql

```

下面我们对这个 Playbook 做一个简单的分析：

第1个任务，我们安装了一些让 Python 能更好地管理 APT 源的辅助库（因为 apt_repository 模块需要借助 python-apt 和 python-pycurl 这两个工具来实现其功能）。

第2个任务，由于 Ubuntu 12.04 中默认的 APT 源是不包含 PHP 5.4.x 及其后续版本的，所以我们安装了包含 PHP 5.5 的 Ondrej 的 PHP5-oldstable 源。

第3个任务，安装了所有 LAMP 服务器需要的软件包，包括运行 Drupal 框架所需的 php5 扩展包。

第4个任务，由于本例基于测试的目的，所以我们禁用了防火墙。如果是在生产环境中，我们需要开启包括 22、80 和 443 号端口在内的必需端口。

第5个任务，启动 LAMP 所需的各项服务，并确保它们开机启动。

4.6.3 配置 Apache

接下来我们将要配置 Apache 服务器，使其能与 Drupal 正常工作。

在 Ubuntu 12.04 中，Apache 默认并未开启 mod_rewrite 功能，通常我们可以使用 sudo a2enmod rewrite 命令来解决这个问题。但是，在 Ansible 中我们可以使用 apache2_module 这个模块来轻松解决这个问题，如代码清单 4-4 所示：

代码清单 4-4 apache2_module 模块的使用

```

- name: Enable Apache rewrite module (required for Drupal).
  apache2_module: name=rewrite state=present
  notify: restart apache

- name: 在 Apache 中为 Drupal 添加 virtualhost
  template:
    src: "templates/drupal.dev.conf.j2"
    dest: "/etc/apache2/sites-available/{{ domain }}.dev.conf"
    owner: root
    group: root
    mode: 0644
  notify: restart apache

- name: 在 sites-enabled 目录中添加 Drupal 所需配置文件的符号链接
  file:
    src: "/etc/apache2/sites-available/{{ domain }}.dev.conf"
    dest: "/etc/apache2/sites-enabled/{{ domain }}.dev.conf"
    state: link
  notify: restart apache

```

第1个任务，使用了 Ansible 的 `apache2_module` 来开启 Apache 的 `rewrite` 功能，其原理是将指定的功能所需的模块做一个符号链接，链接到 `/etc/apache2/mods-enabled` 目录中。

第2个任务，是将一个我们事先定义好的 Jinja2 模块复制到 Apache 的 `sites-available` 目录中，同时设置正确的属主、属组及权限。

第3个任务，是在 `sites-enabled` 目录中添加 Drupal 所需配置文件的符号链接，使得 Apache 能应用这个配置文件。最后，任务还触发了之前定义的 handlers “`restart apache`” 来重新启动 Apache，使得新增加的 Apache 配置文件生效。

下面我们来看一个 Playbook 中所提到的 Jinja2 模块文件 `dru-pal.dev.conf.j2` 的内容。

```
<VirtualHost *:80>
    ServerAdmin webmaster@localhost
    ServerName {{ domain }}.dev
    ServerAlias www.{{ domain }}.dev
    DocumentRoot {{ drupal_core_path }}
    <Directory "{{ drupal_core_path }}">
        Options FollowSymLinks Indexes
        AllowOverride All
    </Directory>
</VirtualHost>
```

可以看到，在标准的 Apache `VirtualHost` 的定义代码中使用了一些 Jinja2 的变量。这种变量的使用方式与我们在 Playbook 中使用变量的方法是一样的，都是两个大括号包含着变量名，比如：`{{variable}}`。

这个模块文件中提到了 3 个变量（`drupal_core_version`，`drupal_core_path`，`domain`），我们在之前提到的变量配置文件 `vars.yml` 中对它们进行定义赋值。

```
---
# 定义 Drupal 要使用的版本（比如：6.x，7.x，8.0.x）
drupal_core_version: "8.0.x"
# 定义 Drupal 将被下载和安装的路径
drupal_core_path: "/var/www/drupal-{{ drupal_core_version }}-dev"
# 定义域名
domain: "drupaltest"
```

当 Ansible 运行到复制模板文件这一步时，模板文件中的 Jinja2 变量将会自动被替换为我们在变量文件 `vars.yml` 中定义的值。

此时，Apache 已经可以启动了。但是，如果此时我们定义的 Drupal 的安装目录还未被创建，通常情况下，Apache 是要报错的。但是在本例中，我们使用的是 handlers 的方式来触发式开启 Apache 服务，也就是说只有在代码清单 4-4 中的 `notify` 前面的 3 个任务都运行成功时，用于启动 Apache 的 handler 才能被触发。

4.6.4 配置 PHP

在之前的章节中，我们曾简要的提到过 Ansible 的 `lineinfile` 模块的使用及案例。`lineinfile`

模块是 Ansible 编辑文件内容的一大利器。对于 PHP 的配置，我们就主要借助 `lineinfile` 模块来完成。

通过编辑修改 PHP 的配置文件，也可以反映出 `lineinfile` 模块在编辑文件方面的简单实用。我们来看下面这个编辑 PHP 配置文件的例子。

```
- name: Enable upload progress via APC.
  lineinfile:
    dest: "/etc/php5/apache2/conf.d/20-apcu.ini"
    regexp: "^apc.rfc1867"
    line: "apc.rfc1867 = 1"
    state: present
    notify: restart apache
```

在这个任务中，我们的目的是开启 PHP 的 `apc.rfc1867` 选项，以确保 APC 支持上传进度条功能。在任务一开始，我们使用 `dest` 关键字告诉 `lineinfile` 模块要被编辑的文件的位置及名称，然后使用正则表达式来搜索 PHP 的配置项 `apc.rfc1867`，接着使用 `line` 关键字来指定被匹配到的行具体应该是什么内容，最后，使用 `state` 关键字来明确指定我们需要这行内容保留下来。


由此可以看出，Ansible 的 `lineinfile` 模块是自动化管理服务配置文件的一大利器。

4.6.5 配置 MySQL

对于本例中的 MySQL 而言，我们首先需要删除系统默认自带的 `test` 数据库，然后为 Drupal 创建一个新库，我们使用下面一段代码实现：

```
- name: 删除 test 数据库
  mysql_db: db=test state=absent
- name: 为 Drupal 创建新库
  mysql_db: "db={{ domain }}" state=present"
```

MySQL 在安装之初都会默认安装一个名为 `test` 的测试数据库，在我们使用 `mysql_secure_installation` 命令来安装 MySQL 时候，也是会提醒我们对其进行删除的。因为这个库对我们的环境没有用处，所以配置 MySQL 的第一步就是删除这个库。接下来，我们创建了一个名为 `{{domain}}` 的数据库，即以 Drupal 网站的域名来命名这个库。

 **提示** Ansible 同时支持包括以下数据库类型在内的大多数数据库类型：MongoDB、MySQL、PostgreSQL、Redis 和 Riak 等。对于 MySQL 而言，Ansible 使用 Python 的 `MySQLdb` 模块（`python-mysqldb`）来对 MySQL 数据库进行管理，并且默认使用 `root` 免密码登录。很显然，这并不符合我们在生产环境中的要求。在实际应用中，我们第一步要做的就是修改 MySQL 的 `root` 密码，并限制 `root` 账号只能本地登录，同时删除不需要的数据库用户。

4.6.6 安装 Drush 和 Composer

Drush 是一个 Drupal 命令行工具，是在 Shell 中操作 Drupal 的桥梁。使用 Drush 不仅可以快速得到网站状态，还可以对网站进行维护，而且还能很方便地做批处理。Composer 是 PHP 的包管理工具。二者是对 PHP 网站管理经常用到的工具。使用如下代码可实现 Drush 和 Composer 的自动化安装：

```
- name: Download Composer installer.
  get_url:
    url: https://getcomposer.org/installer
    dest: /tmp/composer-installer.php
    mode: 0755

- name: Run Composer installer.
  command: >
    php composer-installer.php
    chdir=/tmp
    creates=/usr/local/bin/composer

- name: Move Composer into globally-accessible location.
  shell: >
    mv /tmp/composer.phar /usr/local/bin/composer
    creates=/usr/local/bin/composer
```

前两个任务下载并安装了 Composer，安装之后，在 /tmp 目录下会生成一个 PHP 的应用包 composer.phr，这个包由第 3 个任务移到了目录 /usr/local/bin 下面，并命名为 composer，这样一来，我们就可以直接在命令行中使用 composer 命令来安装 Drush 的依赖关系了。第 2 个任务和第 3 个任务中的 creates 选项是 command 模块和 Shell 模块共有的选项，表示如果文件存在，就不再执行相应的命令。

接下来通过 GitHub 下载并安装 Drush，

```
- name: 从 GitHub 中下载 Drush 代码
  git:
    repo: https://github.com/drush-ops/drush.git
    dest: /opt/drush

- name: 使用 Composer 安装 Drush
  shell: >
    /usr/local/bin/composer install
    chdir=/opt/drush
    creates=/opt/drush/vendor/autoload.php

- name: 创建 Drush 命令的符号链接
  file:
    src: /opt/drush/drush
    dest: /usr/local/bin/drush
    state: link
```

在本例中，我们借助了 Ansible 的 git 模块从 GitHub 上克隆了 Drush 的代码。git 模块只需要再用简单的参数 `repo` 和 `dest` 来分别指定文件在 GitHub 上的 URL 和克隆到本地的存放位置。Drush 代码被下载到 `/opt/drush` 目录后，在任务二中切换到这个目录下，直接利用 `composer` 即可安装。最后在任务三中，为 Drush 的二进制命令文件创建符号链接，这样就能在命令行中直接使用 Drush 命令了。

4.6.7 通过 Git 和 Drush 安装 Drupal

我们将通过 Git 来克隆 Drupal 的代码，并将其保存到之前定义的 Apache 文档根目录 (DocumentRoot) 中，然后使用 Drush 来完成最后的安装。来看下面的代码：

```
- name: 下载 Drupal 代码到 Apache 的 DocumentRoot
  git:
    repo: http://git.drupal.org/project/drupal.git
    version: "{{ drupal_core_version }}"
    dest: "{{ drupal_core_path }}"

- name: 安装 Drupal
  command: >
    drush si -y --site-name="{{ drupal_site_name }}" --account-name=admin
    --account-pass=admin --db-url=mysql://root@localhost/{{ domain }}
    chdir={{ drupal_core_path }}
    creates={{ drupal_core_path }}/sites/default/settings.php
  notify: restart apache

- name: 为 settings.php 设置正确的权限
  file:
    path: "{{ drupal_core_path }}/sites/default/settings.php"
    mode: 0744

- name: 开放 files 目录的所有权限
  file:
    path: "{{ drupal_core_path }}/sites/default/files"
    mode: 0777
    state: directory
    recurse: yes
```

在任务一中，我们从 Drupal 的 Git 源中下载了 Drupal 的代码，下载的版本为我们之前在变量文件中定义的 `drupal_core_version` 值。

在任务二中，我们使用 Drush 的 `si` 命令 (site-install 的缩写) 来安装 Drupal，在安装过程中同时配置了数据库连接，使用 `creates` 来检测重要配置文件 `settings.php` 是否被生成，以此判断 Drupal 是否安装成功，最后触发 handlers 来重启 Apache。

本例中我们用到了一个新变量 `drupal_site_name`，这个变量之前并未在变量文件 `vars.yml` 中定义过，所以我们要在 `vars.yml` 文件中追加如下代码来为该变量赋值：

```
# Drupal 网站的名称
```

```
drupal_site_name: "D8 Test"
```

4.6.8 Drupal 部署过程总结

至此，一个可以自动化安装 LAMP 并安装 Drupal 的 Playbook 已经完成，可以使用 `ansible-playbook` 命令来执行这个 Playbook。

```
ansible-playbook playbook.yml
```

命令执行完成以后，我们可以通过访问 `http://drupaltest.dev`（默认已做好地域名和 IP 地址的映射）来访问你的 Drupal 网站，在首页使用账号密码 `admin/admin` 来登录管理后台。

Drupal 的自动化部署成功的同时，我们也熟悉了 LAMP 的自动化实现，可以将这套配置推而广之，将其应用于 Symfonw、Wordpress、Joomla，以及 Laravel 等的安装。

4.7 实战三：Ansible 部署 Tomcat 企业实战

Apache Solr 是一种高效可扩展的企业级搜索应用服务器。它易于安装和配置，而且附带了一个基于 HTTP 的管理界面。Solr 已经在众多大型的网站中使用，较为成熟和稳定。本节我们将通过部署当前最新版本的 Apache Solr，来向大家详细介绍 Tomcat 8 在 Ubuntu 14.04 系统上的自动化部署。

4.7.1 定义变量并设置 Handlers

与 4.6 节部署 LAMP 相同，我们在 Playbook 开头处先引入用于独立保存和定义变量的变量文件 `vars.yml`，我们的 Playbook 文件依旧命名为 `playbook.yml`。开头内容定义如下：

```
---
- hosts: all
  vars_files:
    - vars.yml
```

在 `playbook.yml` 相同目录下，我们创建变量文件 `vars.yml`，并在 `vars.yml` 中定义如下变量：

```
---
# 软件包下载路径
download_dir: /tmp
# Tomcat 版本号
tomcat_version: 8.0.35
# Tomcat 安装路径
tomcat_dir: /opt/tomcat
# Solr 安装路径
solr_dir: /opt/solr
# Solr 版本号 (最新版)
```



```
solr_version: 6.1.0
```

这 5 个变量分别定义了软件包的下载存放目录、Tomcat 版本号、Tomcat 安装路径、Solr 安装目录、Solr 软件版本号。

接下来定义用于触发式启动 Tomcat 的 handlers，这里我们使用 Ubuntu 上的 Upstart 脚本来管理 Tomcat。

```
handlers:
  - name: start tomcat
command: >
initctlstart tomcat
```

Ubuntu 从 6.10 开始逐步用 Upstart 代替原来的 sysinit，进行服务进程的管理。系统一般默认自带 Upstart，不需要单独安装，我们用来管理 Tomcat 的 Upstart 脚本文件为 tomcat.conf，需放置在被管理主机的 /etc/init 目录下面。其内容如下：

```
description "Tomcat Server"

start on runlevel [2345]
stop on runlevel [!2345]
respawn
respawn limit 10 5

setuid tomcat
setgid tomcat

env JAVA_HOME=/opt/java
env CATALINA_HOME=/opt/tomcat

# Modify these options as needed
env JAVA_OPTS="-Djava.awt.headless=true -Djava.security.egd=file:/dev/./
urandom"
env CATALINA_OPTS="-Xms512M -Xmx1024M -server -XX:+UseParallelGC"

exec $CATALINA_HOME/bin/catalina.sh run

# cleanup temp directory after stop
post-stop script
rm -rf $CATALINA_HOME/temp/*
end script
```

我们将会在接下来的 Playbook 任务中，通过 copy 模块将其发送到目标主机的 /etc/init 目录下。

4.7.2 安装 Java

新版的 Solr 需要 Tomcat 8 以及 JDK 8 以上版本的支持。Ubuntu 14.04 中，APT 源上面默认的是 jdk 7，所以我们不能直接使用 apt 模块来进行 JDK 的安装，需要从 Oracle 官网下

载最新 JDK 8 进行解压安装。考虑到国内连接 Oracle 官网的速度，我们推荐先将 JDK 软件下载到 Ansible 服务器上，再用 copy 模块发送到目标远程主机。

tasks:

- name: 发送 JDK 软件包和 Java 配置文件到远程主机
 - copy: "src={{ item.src }} dest={{ item.dest }}"
 - with_items:
 - src: "./jdk-8u11-linux-x64.tar.gz"
 - dest: "/tmp/"
 - src: "./java.sh"
 - dest: "/etc/profile.d/"
- name: 创建 Java 安装目录
 - command: >
 - mkdir -p /opt/java
- name: 解压 JDK 软件包
 - command: >
 - tar -C /opt/java -xvf {{ download_dir }}/jdk-8u11-linux-x64.tar.gz
 - strip-components=1
- name: 为 Java 命令更新 alternatives
 - command: >
 - update-alternatives --install /usr/bin/java java /opt/java/bin/java 300
- name: 为 javac 更新 alternatives
 - command: >
 - update-alternatives --install /usr/bin/javac javac /opt/java/bin/javac 300

在这部分任务中，任务一中用到的文件 java.sh 用于定义 Java 运行所需的环境变更，其内容如下：

```
export JAVA_HOME="/opt/java"
export CLASSPATH=$JAVA_HOME/lib:$JAVA_HOME/jre/lib
export JRE_HOME=${JAVA_HOME}/jre
export PATH=$PATH:$JAVA_HOME/bin
```

最后两个任务，我们使用 update-alternatives 命令来切换 Java 命令和 javac 命令的版本，这里用到的新版 Java 所用路径就是由 java.sh 文件来定义的。

4.7.3 安装 Tomcat 8

在 Ubuntu 系统中安装 Tomcat 7 非常简单，因为在 Ubuntu 默认的 APT 源中就有 Tomcat 7 的安装包，只需使用 apt 模块就能轻松完成。但是现在 Tomcat 8 已成为主流，而且我们将要安装的新版 Solr 也明确要求 Tomcat 版本要尽量新的，所以接下来就实现 Tomcat 8 的安装。

- name: 创建 Tomcat 安装目录
 - command: >

```

    mkdir -p {{ tomcat_dir }}

- name: 添加运行 Tomcat 所需的普通用户 tomcat
  user: "name=tomcat shell=/sbin/nologin"

- name: 下载 Tomcat 软件包
  get_url:
    url: "http://apache.fayea.com/tomcat/tomcat-8/v{{ tomcat_version }}/bin/
    apache-tomcat-{{ tomcat_version }}.tar.gz"
    dest: "{{ download_dir }}/apache-tomcat-{{ tomcat_version }}.tar.gz"

- name: 解压缩 Tomcat 软件包
  command: >
    tar -C {{ tomcat_dir }} -xvf {{ download_dir }}/apache-tomcat-{{ tomcat_
    version }}.tar.gz --strip-components=1
  creates={{ tomcat_dir }}/conf/server.xml

- name: 发送 Tomcat 的 Upstart 配置文件到远程主机
  copy: "src=./tomcat.conf dest=/etc/init/tomcat.conf"

- name: 重载 Upstart 配置文件
  command: initctl reload-configuration

```

在任务三中，我们选择了官方给出的下载速度最快的链接进行直接下载，在应用中可以根据自己的网络环境选择直接从链接下载或者像 JDK 一样，先下载到 Ansible 服务器本地，然后通过内网传送到目标主机。

在任务五中，我们将 Tomcat 的 Upstart 配置文件发送到了远程目标主机，并在接下来的任务中使 Upstart 重载了它的配置文件，从而可以顺利地管理 Tomcat。

4.7.4 安装 Apache Solr

Ubuntu 系统中默认 APT 源中包含 Apache Solr 的安装包，但是其版本太过陈旧。我们将下载最新版 Solr 进行解压安装，代码如下：

```

- name: 下载新版 Solr 软件包
  get_url:
    url: "http://apache.fayea.com/lucene/solr/{{ solr_version }}/solr-{{ solr_
    version }}.tgz"
    dest: "{{ download_dir }}/solr-{{ solr_version }}.tgz"

- name: 创建 Solr 安装目录
  command: >
    mkdir -p {{ solr_dir }}

- name: 解压缩 Solr 软件包到安装目录
  command: >
    tar -C {{ solr_dir }} -xvzf {{ download_dir }}/solr-{{ solr_version }}.tgz
    --strip-components=1

```

```
creates={{ solr_dir }}/dist/solr-core-{{ solr_version }}.jar
```

在解压缩的任务中，我们使用了 `command` 模块的 `creates` 选项来判断解压后的某个指定文件是否存在，并确定指定版本的 Solr 软件包是否已经被解压过。如果该文件存在，则说明 Solr 软件已经被解压过，则不需再执行解压任务，以减少重复任务，提高效率，同时保证了幂等性。

Solr 软件的安装方法和 Tomcat 很像，解压后不需要编译，直接将软件包放在指定的位置并对配置文件稍做修改即可。

```
- name: 将 Solr 文件部署到指定 Tomcat 目录中
  shell: >
    rsync -av {{ item.src }} {{ item.dest }}
    creates={{ itemcreates }}
    with_items:
      - src: "{{ solr_dir }}/server/solr-webapp/webapp/*"
    dest: "{{ tomcat_dir }}/webapps/solr"
    creates: "/opt/tomcat/webapps/solr/index.html"

      - src: "{{ solr_dir }}/server/lib/ext/*"
    dest: "{{ tomcat_dir }}/webapps/solr/WEB-INF/lib/"
    creates: "/opt/tomcat/webapps/solr/WEB-INF/lib/slf4j-api-1.7.7.jar"

      - src: "{{ solr_dir }}/server/resources/log4j.properties"
    dest: "{{ tomcat_dir }}/webapps/solr/WEB-INF/classes/"
    creates: "/opt/tomcat/webapps/solr/WEB-INF/classes/log4j.properties"
```

上面这个任务将 Tomcat 所需的 Solr 的页面文件和库文件全部部署到位，同样使用 `shell` 模块的 `creates` 选项来预先判断文件是否已经存在，以减少重复操作。

下一步，我们需要修改 Solr 的配置文件 `/path/to/tomcat/webapps/solr/WEB-INF/web.xml`，修改其中的 `<env-entry>` 部分，来指定 Solr 的 Home 目录。修改内容如下：

```
<env-entry>
<env-entry-name>solr/home</env-entry-name>
<env-entry-value>/opt/solr/server/solr</env-entry-value>
<env-entry-type>java.lang.String</env-entry-type>
</env-entry>
```

我们选择将该文件在本地手动配置好后直接发送至目标主机。

```
- name: 覆盖 Solr 配置文件
  copy: "src=./web.xml dest=/opt/tomcat/webapps/solr/WEB-INF/web.xml"
```

接下来，将 Solr 和 Tomcat 安装目录中的所有文件的属主和属组全部设为前面创建的普通用户 `tomcat`，之后就可以触发 `Handlers` 来启动 Tomcat 了。

```
- name: 修改 Solr 和 Tomcat 安装目录下的文件权限，并启动 Tomcat
  file:
```



```

path: "/opt"
owner: tomcat
group: tomcat
recurse: yes
notify: start tomcat

```

在命令行中执行 `ansible-playbook playbook.yml`, Solr 将会自动部署到目标主机。执行完成后, 在浏览器中选择文件 `http://solr.exmpmle.com:8080/solr/index.html` (假设你的目标主机的域名为 `solr.example.com`), 就可以看到 Solr 的管理页面了。

4.8 本章小结

现在, 你应该对 Ansible 的工作场景有了一个比较清楚的了解了。Playbook 是 Ansible 发挥作用的无可争议的核心。Ansible 通过 Playbook 来实现它的具体功能, 同时 Playbook 中的各项任务还可以相同的模块和相近的语法在在命令行中一一实现。

掌握了 Playbook 的用法之后, 接下来的章节我们将深入讨论 Playbook 的高级用法, 包括任务组织、条件判断以及高级变量等。再深入学习, 我们还借助 `role` 来组织 Playbook, 从而大大增加 Ansible 执行任务的灵活性, 并且可以在配置服务器组织架构方面节省大量时间。

Ansible Playbook 拓展

第 4 章当中，我们用到过的 Playbook 及一些功能相对简单的 Playbook 的组合，其实已经包含了日常生产环境中的很多场景。但是，当我们把视野扩展到整个系统管理，还是有许多多的 Ansible 功能需要学习和掌握。

5.1 Handlers

在 4.6 节 LAMP 实战中，我们就已经使用了 Handlers 来实现了重启 Apache 的功能，该实例中，一些修改 Apache 配置文件的操作使用 `notify: restart apache` 触发 Handlers，从而实现了 Apache 的重启。

```
handlers:
  - name: restart apache
    service: name=apache2 state=restarted
tasks:
  - name: 开启 Apache rewrite 模块
    apache2_module: name=rewrite state=present
    notify: restart apache
```

在某些情况下，你可能需要同时调用多个 Handlers，或者需要使用 Handlers 调用其他 Handlers，Ansible 可以很简便地实现这些功能。

下面的例子中，实现了一个任务同时调用多个 Handlers。

```
- name: Rebuild application configuration.
  command: /opt/app/rebuild.sh
```

```

notify:
  - restart apache
  - restart memcached

```

若要实现 Handlers 调用 Handlers，则直接在 Handlers 中使用 notify 选项就可以了，如以下代码所示。

```

handlers:
  - name: restart apache
    service: name=apache2 state=restarted
    notify: restart memcached

  - name: restart memcached
    service: name=memcached state=restarted

```

在使用 Handlers 的过程中，有以下几点需要格外注意。

- Handlers 只有在其所在的任务被执行时，才会被运行；如果一个任务中定义了 notify 调用 Handlers，但是由于条件判断等原因，该任务未被执行，那么 Handlers 同样不会被执行。
- Handlers 只会在 Play 的末尾运行一次；如果想在中间运行 Handlers，则需要使用 meta 模块来实现，例如：`- meta: flush_handlers`。
- 如果一个 Play 在运行到调用 Handlers 的语句之前失败了，那么这个 Handlers 将不会被执行。我们可以使用 meta 模块的 `--force-handlers` 选项来强制执行 Handlers，即使是 Handlers 所在的 Play 中途运行失败也能执行。

5.2 环境变量

在 Ansible 中设置和使用环境变量的方法多种多样。例如，如果我们想为连接远程主机的账号设置一些环境变量，我们可以使用 `lineinfile` 模块直接修改远程用户的 `~/.bash_profile` 文件，如下代码所示：

```

- name: 为远程主机上的用户指定环境变量
  lineinfile: dest=~/.bash_profile regexp=^ENV_VAR= line=ENV_VAR=value

```

在此之后的所有任务都可以使用这些变量。

再如，为了再后续的任务中使用此前定义过的变量，可以使用 `register` 选项来将环境变量存储到自定义的变量中去。

```

- name: 为远程主机上的用户指定环境变量
  lineinfile: dest=~/.bash_profile regexp=^ENV_VAR= line=ENV_VAR=value

- name: 获取刚刚指定的环境变量，并将其保存到自定义变量 foo 中
  shell: 'source ~/.bash_profile && echo $ENV_VAR'
  register: foo

```

```
- name: 打印出环境变量
  debug: msg="The variable is {{ foo.stdout }}"
```

我们在第 4 行使用了“`source ~/.bash_profile`”命令重读了环境变量配置文件，这样就能确保我们接下来获取的是最新生效的环境变量。在某些情况下，若所有任务都运行在一个持久的或准高速缓存的 SSH 会话上的话，如果不重读环境变量配置文件，那么我们所定义的新环境变量 `ENV_VAR` 可能就不会生效。



提示 为什么要使用 `~/.bash_profile` 文件来定义环境变量？其实有很多不同的系统文件可以用来保存环境变量，比如用户家目录下的 `.bashrc`、`.profile`，以及我们所用到的 `.bash_profile`。

Linux 同样也使用文件 `/etc/environment` 来读取环境变量，所以我们可以使用如下方法来指定远程主机上用户的环境变量。

```
- name: Add a global environment variable.
  lineinfile: dest=/etc/environment regexp="^ENV_VAR=" line=ENV_VAR=value
  sudo: yes
```

`lineinfile` 模块可以很方便地处理对环境变量设定量较少的情况。当我们需求大量的环境变量设定的时候，`copy` 模块和接下来要讲的模板将会是不错的选择。

预定义环境变量

对于某一个 Play 来说，我们可以使用 `environment` 选项来为其设置单独的环境变量。比如，我们现在需要为一个下载任务设置 `http` 代理。最简单的情况，我们可以这样实现：

```
- name: 使用指定的代理服务器下载文件
  get_url: url=http://www.example.com/file.tar.gz dest=~/.Downloads/
  environment:
    http_proxy: http://example-proxy:80/
```

但是，一旦任务数量增多或者需要其他环境变量时，这种方法就会变得非常笨重。在此例中，我们可以使用 `playbook` 中的 `var` 区块（或者一个包含变量的外部文件）来传递多个环境变量到 `play` 中，如以下代码所示：

```
vars:
  var_proxy:
    http_proxy: http://example-proxy:80/
    https_proxy: https://example-proxy:443/
    [etc...]
tasks:
  - name: 使用指定的代理服务器下载文件
    get_url: url=http://www.example.com/file.tar.gz dest=~/.Downloads/
```




```
environment: var_proxy
```

如果要为整个系统设置代理服务器，那么建议使用 `/etc/environment` 文件进行定义，如下代码所示：

```
vars:
  proxy_state: present
task:
  - name: Configure the proxy.
    lineinfile:
      dest: /etc/environment
      regexp: "{{ item.regexp }}"
      line: "{{ item.line }}"
      state: "{{ proxy_state }}"
      with_items:
        - { regexp: "^http_proxy=", line: "http_proxy=http://example-proxy:80/" }
        - { regexp: "^https_proxy=", line: "https_proxy=https://example-proxy:443/" }
        - { regexp: "^ftp_proxy=", line: "ftp_proxy=http://example-proxy:80/" }
```

采用这种方法，无论我们使用何种代理协议（http、https 或 ftp），都可以通过变量 `proxy_stat` 来决定是否启用代理服务。当然，我也可以使用类似的方法来设置其他类似系统级别的变量。

 **提示** 我们可以使用如下命令来检测我们在远程主机设置的环境变量是否生效：

```
ansible test -m shell -a 'echo $TEST'
```

注意保持单引号和双引号前后一致。

5.3 变量

Ansible 中变量的命名规则与其他语言或系统中变量的命名规则非常相似。在 Ansible 中，变量以英文大小写字母开头，中间可以包含下划线（`_`）和数字。

合法的变量定义格式如：`foo`、`foo_bar`、`foo_bar_5`、`fooBar`，但是通常我们建议字母都用小写，避免变量名中大小写字母混合的“驼峰式”写法，同时，应尽量避免在变量中间出现数字，尽量让数字出现在变量名末尾。

不合规的变量举例如下：`_foo`、`foo-bar`、`5_foo_bar`、`foo.bar`、`foo bar`。

在 Inventory 文件中，比如 Ansible 的 Hosts 文件，我们使用等号“`=`”来为变量赋值，如：

```
foo=bar
```

在 Playbook 和包含变量设置的配置文件中，我们使用冒号“`:`”来为变量赋值，如：

```
foo: bar
```

5.3.1 Playbook 变量

Ansible 中有多种不同的途径来定义变量。

比如在运行 Playbook 时, 使用 `--extra-vars` 选项来指定额外的变量。

```
ansible-playbook example.yml --extra-vars "foo=bar"
```

我们也可以直接引用 JSON 或 YML 代码来设置额外变量, 或者直接将定义变量的 JSON 或 YAML 代码写入一个文件中, 然后调用这个文件。比如:

```
ansible-playbook example.yml --extra-vars "@even_more_vars.json"
ansible-playbook example.yml --extra-vars "@even_more_vars.yml"
```

上面是 Ad-Hoc 方式中设置额外变量的方法。下面我们来举例说明一下在 Playbook 是怎么设置变量的。

在 Playbook 中, 最常见的定义变量的方法是使用 `vars` 代码块。如 Playbook 内容如下:

```
---
- hosts: example
  vars:
    foo: bar
  tasks:
    # Prints "Variable 'foo' is set to bar".
    - debug: msg="Variable 'foo' is set to {{ foo }}"
```

同时, 变量的定义也可以在一个独立的文件中完成, 当要使用时, 在 Playbook 中使用 `vars_files` 代码块来引用这个文件, 来看个例子。

Playbook 文件内容如下:

```
---
- hosts: example
  vars_files:
    - vars.yml
  tasks:
    - debug: msg="Variable 'foo' is set to {{ foo }}"
```

定义变量的独立文件 `vars.yml` 的内容如下:

```
---
foo: bar
```

上例中使用了 `vars_file` 代码块来调用独立文件 `vars.yml`, 并且能成功读取其中定义的变量。



注意 有读者可能已经注意到, 在 `vars.yml` 文件中, 定义变量的代码是顶格写的。这就是当变量被独立出来定义时的一个特殊之处, 即当在独立文件中定义变量时, 变量可在 YML 中顶格进行定义, 也不需要 `vars` 的标识。

利用 Ansible 的内置环境变量（即使用 `setup` 模块可以查看到的变量），我们还可以实现变量配置文件的有条件导入。

我们来看以下的应用场景：现在生产环境中有多台主机，分别安装了 CentOS 系统和 Debian 系统，同时我们为两套系统设置了两个变量定义文件：`apache_CentOS.yml` 和 `apache_default.yml`，里面同时定义了同一个变量 `apache`，在 `apache_CentOS.yml` 文件中定义为 `apache : httpd`，在 `apache_default.yml` 文件中定义为 `apache : apache2`，这样我们就实现了同一个 Playbook 可以针对不同系统环境实施不同的操作的效果。Playbook 内容如下：

```
---
- hosts: example
  vars_files:
  - [ "apache_{{ ansible_os_family }}.yml", "apache_default.yml" ]
  tasks:
    - service: name={{ apache }} state=running
```

在执行 Playbook 的过程中，Ansible 会主动读取远程主机的 `Factor` 信息，从而获取远程主机的 `ansible_os_family` 的值，并在 `vars_files` 代码块读取该值得到对应名称的变量定义文件，如果没有匹配到合适的文件名，将默认读取 `apache_default.yml` 中的设定。

5.3.2 在 Inventory 文件中定义变量

在 Ansible 中，Inventory 文件通常是指 Ansible 的主机和组的定义文件 `Hosts`（默认路径为 `/etc/ansible/hosts`，简称 `Hosts` 文件）。在 `Hosts` 文件中，变量会被定义在主机名的后面或组名的下方，如下面这个例子所示：

```
# 为某台主机指定变量，作用范围仅限于该台主机
[shanghai]
app1.example.com proxy_state=present
app2.example.com proxy_state=absent

# 为主机组指定变量，作用范围为整个主机组
[shanghai:vars]
cdn_host=sh.static.example.com
api_version=3.0.
```

在 Inventory 文件中直接定义变量方法虽然简单直观，但当所需要定义的变量多，并且在被多台主机同时应用的时候，这种方法就会显得非常麻烦。而且，事实上，Ansible 的官方手册中也并不建议人们把变量直接定义在 `Hosts` 文件中。

在执行 Ansible 命令时，Ansible 默认会从 `/etc/ansible/host_vars/` 和 `/etc/ansible/group_vars/` 两个目录下读取变量定义，如果 `/etc/ansible` 下面没有这两个目录，可以直接手动创建，并且可以在这两个目录中创建与 `Hosts` 文件中主机名或组名同名的文件来定义变量。

举例来说，我们现在要给主机 `app1.example.com` 设置一组变量，那就可以直接在 `/etc/ansible/host_vars/` 目录下创建一个名为 `app1.example.com` 的空白文件，然后在文件中以

YAML 语法来定义所需的变量，如以下代码所示：

```
---
foo: bar
baz: qux
```

如此一来，变量 `foo` 和 `baz` 将自动定义给主机 `app1.example.com`。

同理，要想针对整个 `shanghai` 主机组定义一些变量，则只需在 `/etc/ansible/group_vars/` 目录下创建与主机组同名的 YAML 文件来定义变量就可以了。

在 5.3.1 节中，最后一个例子应用变量的方式与这个例子中的方式非常相似，朋友们可以将二者进行比较，以便于理解。

5.3.3 注册变量

注册变量，其实就是将操作的结果，包括标准输出和标准错误输出，保存到变量中，然后再根据这个变量的内容来决定下一步的操作，在这个过程中用来保存操作结果的变量就叫注册变量。我们在 Playbook 中使用 `register` 来声明一个变量为注册变量。

在第 4 章中，我们就曾使用 `register` 来声明注册变量来保存命令运行结果，然后用其来判断是否需要启动 Node.js，再来回顾一下那段代码：

```
- name: 获取正在运行的 Node.js app 列表
  command: forever list
  register: forever_list
  changed_when: false

- name: 启动 Node.js app
  command: forever start {{ node_apps_location }}/app/app.js
  when: "forever_list.stdout.find('{{ node_apps_location }}/app/app.js') == -1"
```

在这段代码中，我们使用 Python 内置的字符串的 `find` 方法来查找 `app.js` 的路径，如果没找到，程序就会自动启动 Node.js。

我们将会在第 5.4.2 节继续深入讨论 `register` 的更多用法。

5.3.4 使用高阶变量

对于普通变量，例如由 Ansible 命令行设定的、在 Hosts 文件中定义的，再或者在 Playbook 和变量定义文件中定义的，这些变量都被称为简单变量或普通变量，我们可以直接在 Playbook 中使用双大括号加变量名来读取变量内容，形如 `{{variable}}`。比如下面的例子：

```
- command: /opt/my-app/rebuild {{ my_environment }}
```

当 Playbook 运行这段命令时，变量 `my_environment` 将会自动被替换为其所对应的变量内容。

Ansible 中除了这些普通变量之外，还有数组变量或者叫列表变量。由于 Ansible 是基于

Python 语言开发的，所以我们这里就称之为列表。列表的定义方法如下：

```
foo_list:
  - one
  - two
  - three
```

列表定义完成后，要读取其中第一个变量，有以下两种方法：

```
foo[0]
foo|first
```

像 `foo[0]` 这种读取列表变量的方法是典型的 Python 语法格式，0 表示第 1 个元素，1 表示第 2 个元素，依此类推。第 2 种方法 `foo|first` 则采用的是 Jinja2 语法，这种读取变量的方法相对较麻烦，且使用频率不高，这里不再进行深入讲解。推荐大家使用第 1 种读取变量方法。

接下来我们将介绍另外一种更为复杂的变量，它类似于 Python 中字典的概念，但比字典的维度要高，更像是二维字典。Ansible 内置变量 `ansible_eth0` 就是这样一种变量，它用来保存远程主机上面 `eth0` 接口的信息，包括 IP 地址和子网掩码等。下面我们使用 `debug` 模块来展示一下变量 `ansible_eth0` 的内容。

```
tasks:
  - debug: var=ansible_eth0
```

运行包含该内容的 Playbook 后，这段代码对应的输出结果如下所示：

```
TASK: [debug var=ansible_eth0] *****
ok: [webserver] => {
  "ansible_eth0": {
    "active": true,
    "device": "eth0",
    "ipv4": {
      "address": "10.0.2.15",
      "netmask": "255.255.255.0",
      "network": "10.0.2.0"
    },
    "ipv6": [
      {
        "address": "fe80::a00:27ff:feb1:589a",
        "prefix": "64",
        "scope": "link"
      }
    ],
    "macaddress": "08:00:27:b1:58:9a",
    "module": "e1000",
    "mtu": 1500,
    "promisc": false,
    "type": "ether"
  }
}
```

我们可以看到，这段输出中，从第 3 行开始，都是变量 `ansible_eth0` 的内容，可以说是一个内容庞大的变量了。当我们想要读取其 IPv4 地址时，可使用如下两种方法实现：

```
{{ ansible_eth0.ipv4.address }}
{{ ansible_eth0['ipv4']['address'] }}
```

由此我们可以看出，Ansible 中多级变量的调用，使用中括号和点号都是可以的。

5.3.5 主机变量和组变量

Ansible 为用户提供了用于批量定义主机的管理文件，即 Hosts 文件，默认存放位置是 `/etc/ansible/hosts`。有了这个文件，我们可以非常便捷地在里面为主机分组，极大地简化了多主机的操作。

在 Hosts 文件中，我们使用如下格式定义主机组：

```
[group]
host1
host2
```

为每个主机定义自己专属变量最直接、最简单的方法就是：在 Hosts 文件中，在对应主机名的后面直接定义。如下所示：

```
[group]
host1 admin_user=jane
host2 admin_user=jack
host3
```

这样我们就为 `host1` 和 `host2` 分别定义的一个变量，`host3` 主机则无法使用该变量。

如果要对整个主机组设置变量，则采用如下方法：

```
[group:vars]
admin_user=john
```

这样一来，变量将会对主机组 `group` 下面的所有主机生效，就相当于给其下的每一台主机分别定义了一次变量 `admin_user`。

以上定义主机变量和主机组变量的方法，在主机或主机组数量较少的情况下非常方便有效。但当我们要为非常多的主机和主机组分别设置不同的变量时，这种方法就会显得比较笨拙。

1. group_vars 和 host_vars

Ansible 在运行任务前，都会搜索与 Hosts 文件同一个目录下的两个用于定义变量的目录：`group_vars` 和 `host_vars`。

我们可以在这两个目录下放一些使用 YAML 语法编辑的定义变量的文件，并以对应的主机名和主机组名来命名这些文件，这样在运行 Ansible 时，Ansible 会自动去这两个目录下

读取针对不同主机和主机组的变量定义。我们可以通过下面的例子来加深一下理解。

1) 对主机组 group 设置变量。

```
---
# File: /etc/ansible/group_vars/group
admin_user: john
```

2) 对主机 host1 设置变量。

```
---
# File: /etc/ansible/host_vars/host1
admin_user: jane
```

除此之外，我们还可以在 `group_vars` 和 `host_vars` 两个文件夹下定义 `all` 文件，来一次性地为所有的主机组和主机定义变量。

2. 巧妙使用主机变量和组变量

有些时候，我们在运行 Ansible 任务时，可能需要从一台远程主机上获取另一台远程主机的变量信息，有一个神奇的变量 `hostvars` 可以帮我们实现这一需求。变量 `hostvars` 包含了指定主机上所定义的所有变量。

比如，我们想获取 `host1` 上的变量 `admin_user` 的内容，在任意主机上直接使用下面这行代码即可。

```
{{ hostvars['host1']['admin_user'] }}
```

Ansible 提供了一些非常有用的内置变量，这里我们列举几个常用的。

- ❑ `groups`: 包含了所有 `Hosts` 文件里主机组的一个列表。
- ❑ `group_names`: 包含了当前主机所在的所有主机组名的一个列表。
- ❑ `inventory_hostname`: 通过 `Hosts` 文件定义主机的主机名（与 `ansible_home` 不一定相同）。
- ❑ `inventory_hostname_short`: 变量 `inventory_hostname` 的第 1 部分，比如 `inventory_hostname` 的值是 `books.ansible.com`，那么 `inventory_hostname_short` 就是 `books`。
- ❑ `play_hosts`: 将执行当前任务的所有主机。

5.3.6 Facts (收集系统信息)

1. Facts 信息

在运行任何一个 Playbook 之前，Ansible 默认会先抓取 Playbook 中所指定的所有主机的系统信息，这些信息我们称之为 `Facts`。或许你已经注意到，在之前我们运行的所有 Playbook 任务中，都会出现类似下面代码的内容：

```
ansible-playbook playbook.yml
```

上述命令的运行结果如下：

```
PLAY [group] *****
GATHERING FACTS *****
ok: [host1]
ok: [host2]
ok: [host3]
```

Facts 信息包括(但不仅限于)远程主机的 CPU 类型、IP 地址、磁盘空间、操作系统信息以及网络接口信息等, 这些信息对于 Playbook 的运行至关重要。我们可以根据这些信息来决定是否要继续运行下一步任务, 或者将这些信息写入某个配置文件中。

我们可以使用 `setup` 模块来获取对应主机上面的所有可用的 Facts 信息。比如:

```
ansible munin -m setup
```

运行结果如下:

```
munin.midwesternmac.com | success >> {
  "ansible_facts": {
    "ansible_all_ipv4_addresses": [
      "167.88.120.81"
    ],
    "ansible_all_ipv6_addresses": [
      "2604:180::a302:9076",
      ...
    ]
  }
}
```

在某些用不到 Facts 信息的 Playbook 任务中, 我们可以在 Playbook 中设置 `gather_facts: no` 来暂时让 Ansible 在执行 Playbook 任务之前跳过收集远程主机 Facts 信息这一步, 这样可以为任务节省几秒钟的时间, 如果主机数量多的话, 就能节省更多的时间。在 Playbook 中设置 `gather_facts` 的方法如下:

```
- hosts: db
  gather_facts: no
```

在实际应用当中, 运用得比较多的 Facts 变量有 `ansible_os_family`、`ansible_hostname`、`ansible_memtotal_mb` 等, 这些变量通常会被拿来用作 `when` 语句的判断条件, 来决定下一步的操作。



提示 如果远程主机上安装了 `Facter` 或 `Ohai`, 那么 Ansible 将会把这两个软件所生成的 Facts 信息也给收集回来, Facts 变量名分别以 `facter_` 和 `ohai_` 开头进行标示。如果你的环境中还有 `Puppet` 或者 `Chef` 等管理工具, 那么你也可以使用 Ansible 便捷地获取它们的 Facts 信息。但是, 通常情况下, Ansible 自己的 Facts 变量就已经足够满足需求了。

Playbook 在不同的操作系统上获取到的 Facts 变量的格式是不一样的, 这一点在使用的过程务必多加注意。

2. 本地 Facts 变量

下面我们介绍一种在远程主机本地定义 Facts 变量的方法。

我们可以把需要定义的变量写进一个以 .fact 结尾的文件中，这个文件可以是 JSON 文件或 INI 文件，或者是一个可以返回 JSON 代码的可执行文件。然后将其放置在 /etc/ansible/facts.d 文件夹中，Ansible 在执行任务时会自动到这个文件夹下读取变量信息。

比如，我们在远程主机上创建了一个 .fact 文件 /etc/ansible/facts.d/settings.fact，文件内容如下：

```
[users]
admin=jane,john
normal=jim
```

接下来，使用 setup 模块就可以读取到这两个变量，如下所示：

```
ansible hostname -m setup -a "filter=ansible_local"
munin.midwesternmac.com | success >> {
  "ansible_facts": {
    "ansible_local": {
      "settings": {
        "users": {
          "admin": "jane,john",
          "normal": "jim"
        }
      }
    }
  },
  "changed": false
}
```

如果在一个 Playbook 中，只有部分 Playbook 任务用到了远程主机自定义的本地 Facts，那么我们可以使用下面一段代码来明确地指明只显示这些本地 Facts。

```
- name: 重新获取本地 Facts
  setup: filter=ansible_local
```

提示 定义本地 Facts 的方法通常作为一种临时的定义变量的手段，我们还是建议将变量集中定义在 Ansible 服务器端集中管理。但是在某些特殊情况下，比如远程主机的系统环境经常发生变化，我们就需要在 /etc/ansible/facts.d 下使用脚本文件来动态地生成 Facts 变量。其他大部分情况下，都建议将变量以各种形式集中定义在 Ansible 服务器端。

注意 setup 模块的 filter 功能，目前暂不支持 Windows 系统。

5.3.7 Ansible 加密模块 Vault

当我们使用 Ansible 完全自动化地维护我们的服务器的时候，在运行某些任务时，不可避免地会接触到一些密码或其他敏感数据，这些数据有可能是管理员密码、SSH 私钥或远程主机的认证信息。

当我们把这些数据存放在普通的变量文件或 Hosts 文件中时，如果整个项目被复制迁移，这些数据将很容易被其他人接触，造成安全风险。所以，对于这些敏感数据，我们应该特殊对待。

通常我们使用下面两种方法对敏感数据进行管理：

1) 文件密码管理工具，如 HashiCorp 的 Vault 和 Square 的 Keywhiz，或者使用主机提供商的服务，如亚马逊的 Key Management Service (KMS) 和微软 Azure 的 Key Vault。

2) Ansible 自带的 Vault 加密功能，Vault 可以将经过加密的密码和敏感数据同 Playbook 存储在一起。

对于大部分的 Ansible 项目来说，Ansible 自带的 Vault 功能都能满足需求。Ansible Vault 的工作方式与现实生活中的保险柜的工作方法很像。

❑ 我们可以把 Ansible 任务中用到的任意文件放入 Vault 保险柜中。

❑ Ansible Vault 会使用密码来加密这些文件，与用钥匙把保险柜的门锁起来一样。

❑ 我们把密码（钥匙）保存在一个只有我们自己知道或都有权访问的地址，与 Playbook 独立分开存储。

❑ 在我们需要运行 Playbook 的时候，我们拿出密码（钥匙），解密敏感数据（打开保险柜门，拿出数据），就能正常执行 Playbook 任务了。

接下来，我们通过一个实例来了解一下这个过程。下面一段 Playbook 代码使用 API key 的方式来访问一个服务的 API，

```
---
- hosts: appserver
  vars_files:
    - vars/api_key.yml
  tasks:
    - name: Connect to service with our API key.
      command: connect_to_service
      environment:
        SERVICE_API_KEY: "{{ myapp_service_api_key }}"
```

本例中，用于命令验证的 API key 就存储在一个纯文本文件 vars/api_key.yml 中，内容如下：

```
---
myapp_service_api_key: "yJJvPqhggxyPZMispRycaVMBmBWPqYDf3DFanPxAMAm4UZcw"
```

这种将 key 存储在纯文件文件中的做法非常便捷，但是并不安全。如果我们使用 Ansible Tower 和 Jenkins 等工具来运行 Playbook，或者 Playbook 在一个共享的环境中时，这种存储 key 的方法就更不可取。或许我们有非常严格的主机操作和系统安全规范，但是我们并不能

保证每一位开发者或管理员都能严格遵守，人通常是整个环节中最不稳定的因素。

Ansible Vault 可以为我们提供非常高的安全加密级别，这将很好地帮我们解决后顾之忧。使用如下命令，可以利用 Vault 给文件加密：


```
$ ansible-vault encrypt api_key.yml
Vault password:
```

按提示输入加密密码，文件就会被加密。此刻，当我们再次打开文件 `api_key.yml`，会看到下面的内容：

```
1 $ANSIBLE_VAULT;1.1;AES256
2 653635363963663439383865313262396665353063663839616266613737616539303
3 530313663316264336133626266336537616463366465653862366231310a30633064
4 633234306335333739623661633132376235666563653161353239383664613433663
5 1303132303566316232373865356237383539613437653563300a3263386336393866
6 376535646562336664303137346432313563373534373264363835303739366362393
7 639646137656633656630313933323464333563376662643336616534353234663332
8 656138326530366434313161363562333639383864333635333766316161383832383
9 831626166623762643230313436386339373437333830306438653833666364653164
10 6633613132323738633266363437
```

除了 `encrypt` 选项之外，关于 `ansible-vault` 命令有几个比较常用的选项，列举如下。

- `edit`：用于编辑 `ansible-vault` 加密过的文件。
- `rekey`：重新修改已被加密文件的密码。
- `create`：创建一个新文件，并直接对其进行加密。
- `view`：查看经过加密的文件。
- `decrypt`：解密文件。

 **提示** 以上所有选项都可以在后面跟多个文件进行操作，比如：`ansible-vault create x.yml y.yml z.yml`。

除了手动输入密码进行解密以外，Ansible 还提供了以密码文件的形式来解密的认证方式，这类似于 SSH 的密钥认证。SSH 将密钥放于 `~/.ssh` 目录下面，Ansible Vault 将密码文件放置于 `~/.ansible/`，对于这个文件也必须有严格的权限控制，需设置其权限为 600。现在我们可以 `~/.ansible/` 目录下创建一个权限为 600 的纯文本文件 `vault_pass.txt`，并写入我们的 Vault 密码，使用如下命令就可非交互式地使用被加密过的 Playbook 运行任务了。

```
$ ansible-playbook test.yml --vault-password-file ~/.ansible/vault_pass.txt
```

我们也可以使用可执行脚本来生成单行的 Vault 密码，比如 `~/.ansible/vault_pass.py`，前提是该脚本只能生成一行密码数据。

如果系统上通过 `pip install cryptography` 命令安装了 Python 的 `cryptography` 模块，那么

这将会加快 Vault 的运行速度。



提示 你是否对 Ansible Vault 的加密方式感到担心？Ansible Vault 采用 AES-256 加密算法对文件进行加密，这种算法是极其安全的。即使是使用目前世界上运算速度最快的集群来 7×24 小时地解密一个通过该算法加密的文件，也需要数十亿年的时间才能完成破解。所以，安全算法是没有问题的，我们要做的就是保管好自己的 Vault 加密密码。

可以在以下官方文档中参考更多的选项用法及详细案例：http://docs.ansible.com/ansible/playbooks_vault.html。

5.3.8 变量优先级

通过本节对变量的学习，可能会有人问了，定义变量的方式有那么多，我如何才能确定哪一个定义会最终生效呢？下面我们就来学习一下变量优先级的的问题。

如果同样名称的变量在多个地方被定义，那么究竟以哪一次定义的值为准呢？Ansible 官方给出了如下由高到低的优先级排序：

- 1) 在命令行中定义的变量（即用 `-e` 定义的变量）；
- 2) 在 Inventory 中定义的连接变量（比如 `ansible_ssh_user`）；
- 3) 大多数的其他变量（命令行转换、play 中的变量、included 的变量、role 中的变量等）；
- 4) 在 Inventory 定义的其他变量；
- 5) 由系统通过 `gather_facts` 方法发现的 Facts；
- 6) “Role 默认变量”，这个是默认的值，很容易丧失优先权。

通过一段时间的使用之后，大部分人都会有自己的一套定义变量的方法或习惯。下面我们总结一些变量定义方面的小技巧，希望可以给初学者提供一些有益的参考。

- ❑ Role（下章将讲到）中的默认变量应设置得尽可能的合理，因为它优先级最低，以防这些亦是在其他地方都没被定义，而 Role 的默认亦是又定义的不合理而产生问题；
- ❑ Playbook 中应尽量少地定义变量，Playbook 中用的变量应尽量定义在专门的变量文件中，通过 `vars_files` 引用，或定义在 Inventory 文件中；
- ❑ 只有真正与主机或主机组强相关的变量才定义在 Inventory 文件中；
- ❑ 应尽量少地在动态或静态的 Inventory 源文件中定义变量，尤其是不要定义那些很少在 Playbook 中被用到的变量；
- ❑ 应尽量避免在命令行中使用 `-e` 选项来定义变量。只有在我们不用去关心项目的可维护性和任务幂等性的时候，才建议使用这种变量定义方式。比如只是做本地测试，或者运行一个一次性的 Playbook 任务。

5.4 if/then/when——流程控制

条件判断在 Ansible 任务中的使用频率非常高。有些任务使用带有幂等性检查的模块，比如 yum 和 apt 模块可以检测软件包是否已被安装，而在这个过程中我们不用做太多的人工干预。

但是，大部分的 Ansible 任务，尤其是那些需要用户输入内容的 shell 模块和 command 模块任务，需要对用户的输入内容或任务的运行结果进行判断，这个时候流程控制就显得非常重要了。

本节涵盖 Ansible 中可能用到的主要的条件判断语句，这将帮助我们判断一条任务运行结果的状态是成功还是失败。

5.4.1 Jinja2 正则表达、Python 内置函数和逻辑判断

在开始讲解不同的条件判断语句的用法之前，我们先来简单介绍一小部分 Jinja2 的语法（Ansible 在配置模板文件和进行条件判断时都会用到 Jinja2 语法）以及一些 Ansible 也能够使用的 Python 内置函数。

Jinja2 支持的数据类型有：字符串型（如 “strings”）、整数型（如 45）、浮点数值型（如 42.33）、列表（如 [1, 2, 3, 4]）、元组（与列表类型格式一样，只是内容无法修改）、字典（如 {key: value, key2: value2}），还有布尔型（如 true 或 false）。

Jinja2 同时也支持基本的数据运算，如加、减、乘、除和比较（== 表示相等，!= 表示不相等，>= 表示大于等于，等等）。逻辑运算符支持 and（与）、or（或）、not（非），可以使用小括号来对逻辑运算符进行分组使用。

对于有编程经验的人士来说，只需要很短的时候便可对 Jinja2 这些基本语法熟悉起来。

下面我们来看几个 Jinja2 的语法示例。

下列表达式的运算结果都为 'true':

```
1 in [1, 2, 3]
'see' in 'Can you see me?'
foo != bar
(1 < 2) and ('a' not in 'best')
```

下列表达式的运算结果都为 'false':

```
4 in [1, 2, 3]
foo == bar
(foo != foo) or (a in [1, 2, 3])
```

除此之外，Jinja2 还提供了非常有用的 “test” 语句。比如，我们可以使用如下语句来判断变量 foo 是否被定义过。

```
foo is defined
```

当变量 foo 被定义过，那么这个表达式的结果就是 true，相反则为 false。使用起来非常

直接，就与直接用英文对话一样。类似的还有：`undefined`（与 `defined` 相反），`equalto`（与 `==` 等效），`even`（判断对象是否是偶数）以及 `iterable`（判断对象是否可迭代）。后续章节中，我们将会对所有的 `test` 语句进行讲解，在这里我们只需要了解 Ansible 利用 Jinja2 表达式可以实现非常丰富的功能就可以了。

在少数 Jinja2 并不能发挥强大功能的场景中，我们可以使用 Python 的内置方法来进行补充，比如：`string.split` 和 `[number].is_signed()` 等。

我们来看如下一个应用场景，目前我们有一款软件版本号为 4.6.1，现在有一个任务需要通过判断软件的版本号来确定要不要执行接下来的任务，如果主版本号为 4 就执行任务，其他版本则不执行。这时，Jinja2 表达式将不再适合，我们可以通过 Python 的内置方法，使用点号“.”来对版本号进行拆分后取得第 1 位主版本号，然后用它与数字 4 进行比较。具体代码如下：

```
- name: 当软件主版本号为 4 的时候进行操作
  [task here]
  when: software_version.split('.')[0] == '4'
```

通常我们建议尽量使用更为简洁的 Jinja2 语句来进行判断，但是在涉及变量的复杂操作时，Python 的内置方法还是不错的选择。

5.4.2 变量注册器 register

我们在之前的注册变量那节讲过，任何一个任务都可以注册一个变量用来存储其运行结果，该注册变量在随后的任务中将像其他普通变量一样被使用。

大部分情况下，我们使用注册器用来接收 shell 命令的返回结果，结果中包含标准输出（`stdout`）和错误输出（`stderr`）。使用下面一段代码即可调用注册器来获取 shell 命令的返回结果。

```
- shell: my_command_here
  register: my_command_result
```

命令结果获取完成之后，可以使用注册变量的 `stdout` 方法来读取标准输出的内容：`my_command_result.stdout`；使用 `stderr` 方法来读取标准输入的内容：`my_command_result.stderr`。



注意 如果想查看一个注册变量都有哪些属性，那么在运行一个 Playbook 的时候，使用 `-v` 选项来检查 Playbook 的运行结果，通常我们会得到如下 4 种类型的运行结果。

- ☐ **changed**: 任务是否对远程主机造成的变更
- ☐ **delta**: 任务运行所用的时间
- ☐ **stdout**: 正常的输出信息
- ☐ **stderr**: 错误信息

5.4.3 when 条件判断

很多任务只有在特定条件下才能执行，这就是 when 语句发挥作用的地方。比如我们只需要在数据库服务器上安装 MySQL 软件，根据系统种类来确定是使用 apt 模块还是 yum 模块进行软件包管理。

我们来看一个例子，假设我们的所有服务器上都有一个布尔变量 is_db_server，在数据库服务器上，其值为 true，其他主机上值为 false，我们只需要在数据库服务器上安装 MySQL 软件包，

```
- yum: name=mysql-server state=present
  when: is_db_server
```

如果我们只在数据库服务器上定义 is_db_server 变量，其他主机上没有定义这个变量，这里我们就需要增加一个判断条件，来判断变量是否被定义。

```
- yum: name=mysql-server state=present
  when: (is_db_server is defined) and is_db_server
```

当 when 语句和注册变量结合起来的时候，其功能将更为强大。举例来说，我们想检查一个应用的运行状态，并判断返回的状态值，当状态为“ready”时，再执行下一步操作。任务代码如下：

```
- command: my-app --status
  register: myapp_result
- command: do-something-to-my-app
  when: "'ready' in myapp_result.stdout"
```

这个例子稍显刻意，但是它展示了 when 语句和注册变量结合的基本用法。下面我们再看一个实际生产中的例子。

```
# From our Node.js playbook - register a command's output, then see
# if the path to our app is in the output. Start the app if it's
# not present.
# 这是从 Node.js 项目中摘取的一段代码，使用注册器保存命令运行结果，并对其进行判断
- command: forever list
  register: forever_list
- command: forever start /path/to/app/app.js
  when: "forever_list.stdout.find('/path/to/app/app.js') == -1"
# 以下代码取自之前的 Node.js 安装的 Playbook，使用注册变量保存命令的执行结果，然后通过注册变量判断结果中是否包含我们 App 的路径，如果不包含，就启用我们的 App
  when: ping_hosts
# 如果所在的分支不在 git 的分支列表中，就运行 git-cleanup.sh 脚本
- command: chdir=/path/to/project git branch
  register: git_branches
- command: /path/to/project/scripts/git-cleanup.sh
  when: "(is_app_server == true) and ('interesting-branch' not in git_branches.stdout)"
```

```
# 如果当前 PHP 版本为 7.0, 就执行 PHP 降级操作
- shell: php --version
  register: php_version
- shell: yum -y downgrade php*
  when: "'7.0' in php_version.stdout"
# 如果远程主机的 hosts 文件不存在, 就传一个文件 file 过去
- stat: path=/etc/hosts
  register: hosts_file
- copy: src=path/to/local/file dest=/path/to/remote/file
  when: hosts_file.stat.exists == false
```

5.4.4 changed_when、failed_when 条件判断

与 when 语句类似, 我们可以使用 changed_when 语句和 failed_when 语句对命令运行的结果进行判断。

对于 Ansible 来说, 其很难判断一个命令的运行是否符合我们的实际预期, 尤其是当我们使用 command 模块和 shell 模块时, 如果不使用 changed_when 语句, Ansible 将永远返回 changed。大部分模块都能正确返回运行结果是否对目标主机产生影响, 我们依然可以使用 changed_when 语句来对返回信息进行重写, 根据任务返回结果来判定任务的运行结果是否真正符合我们预期。

正常情况下, 当我们使用 PHP Composer 来安装项目依赖项的时候, 无论是否安装或升级的某些软件, Ansible 任务的返回结果都是 changed。但是当我们使用 changed_when 语句, 并结合注册变量对任务返回结果进行判断后, 再来决定是否显示状态为 changed, 将更加符合我们的实际需求。比如:

```
- name: Install dependencies via Composer.
  command: "/usr/local/bin/composer global require phpunit/phpunit --prefer-dist"
  register: composer
  changed_when: "'Nothing to install or update' not in composer.stdout"
```

由此我们可以看出, 当 PHP Composer 安装或升级了某些软件的时候, 也就是其运行结果中不包含 “Nothing to install or update” 字段的时候, Ansible 才会返回运行状态为 changed, 这更符合我们的需要。

有一些命令会将自己的运行结果写入标准错误输出 stderr 中, 而不是通常的标准输出 stdout 中, 这时可以使用 failed_when 来对结果进行判断, 从而告诉 Ansible 真正的运行结果到底是成功还是失败。

在下面的例子中, 我们将通过判断 Jenkins CLI 命令的错误输出来判定命令是否真的运行失败。代码如下:

```
- name: 通过 CLI 导入 Jenkins 任务
  shell: >
    java -jar /opt/jenkins-cli.jar -s http://localhost:8080/
    create-job "My Job" < /usr/local/my-job.xml
```



```
register: import
failed_when: "import.stderr and 'already exists' not in import.stderr"
```

本例我们希望当命令返回错误信息并且返回的错误信息中不包含“already exists”的内容时，再通知 Ansible 显示命令运行失败。像本例中这种“already exists”的返回信息到底是应该写入 stdout 还是 stderr 中，目前还存在争议，但是有了 failed_when 语句，这对 Ansible 用户来说都一样。

5.4.5 ignore_errors 条件判断

在有些情况下，一些必须运行的命令或脚本会报一些错误，而这些错误并不一定真的说明有问题，但是经常会给接下来要运行的任务造成困扰，甚至直接导致 Playbook 运行中断。

这时候，我们可以在相关任务中添加 ignore_errors: true 来屏蔽所有错误信息，Ansible 也将视该任务运行成功，不再报错，这样就不会对接下来要运行的任务造成额外困扰。但是要注意的是，我们不应过度依赖 ignore_errors，因为它会隐藏所有的报错信息，而应该把精力集中在寻找报错的原因上面，这样才能从根本上解决问题。

5.5 任务间流程控制

5.4 节就任务内部流程的控制进行了详细的讲解。接下来，我们将向大家介绍任务间流程控制的方法。

5.5.1 任务委托

默认情况下，Ansible 的所有任务都是在我们指定的机器上面运行的，当在一个独立的群集环境中配置时，这并没有什么问题。而在有些情况下，比如给某台服务器发送通知或向监控服务器中添加被监控主机，这个时候任务就需要在特定的主机上运行，而非一开始指定的所有主机。此时就需要用到 Ansible 的任务委托功能。

使用 delegate_to 关键字便可以配置任务在指定的机器上执行，而其他任务还是在 hosts 关键字配置的所有机器上运行，当到了这个关键字所在的任务时，就使用委托的机器运行。而 facts 还适用于当前的 host，下面我们演示一个例子，使用 Munin 在监控服务器中添加一个被监控主机。

```
---
- hosts: webserver
  tasks:
    - name: Add server to Munin monitoring configuration.
      command: monitor-server webserver {{ inventory_hostname }}
      delegate_to: "{{ monitoring_master }}"
```

由本例可以看出，我们虽然在 Playbook 开头指定了操作对象是所有 webserver，但是添

加监控对象这一任务却只需要在监控服务器上运行，所以我们就使用了 `delegate_to` 来指定运行此任务的主机。

如果我们想将一个任务在 Ansible 服务器本地运行，除了将任务委托给 127.0.0.1 之外，还可以全用 `local_action` 方法来完成。看下面两个功能一模一样的例子：

```
- name: Remove server from load balancer.
  command: remove-from-lb {{ inventory_hostname }}
  delegate_to: 127.0.0.1

- name: Remove server from load balancer.
  local_action: command remove-from-lb {{ inventory_hostname }}
```

将上述两个功能一样的案例进行比较记忆，加深印象。

5.5.2 任务暂停

在有些情况下，一些任务的运行需要等待一些状态的恢复，比如某一台主机或者应用刚刚重启，我们需要等待它上面的某个端口开启，此时我们就不得不将正在运行的任务暂停，直到其状态满足我们需求。先来看下面的例子：

```
- name: Wait for webserver to start.
  local_action:
    module: wait_for
    host: webserver1
    port: 80
    delay: 10
    timeout: 300
    state: started
```

本例中我们结合前面所讲的 `local_action` 方法和 `wait_for` 模块来完成了任务的暂停操作。这个任务将会每 10s 检查一次主机 `webserver1` 上面的 80 端口是否开启，如果超过 300s，80 端口仍未开启，将会返回失败信息。

总结一下，Ansible 的 `wait_for` 模块常用于如下一些场景中：

- ❑ 使用选项 `host`、`port`、`timeout` 的组合来判断一段时间内主机的端口是否可用；
- ❑ 使用 `path` 选项（可结合 `search_regx` 选项进行正则匹配）和 `timeout` 选项来判断某个路径下的文件是否存在；
- ❑ 使用选项 `host`、`port` 和 `stat` 选项的 `drained` 值来判断一个给定商品的活动连接数是否被耗尽；
- ❑ 使用 `delay` 选项来指定在 `timeout` 时间内进行检测的时间间隔，时间单位为秒。


5.6 交互式提示

在少数情况下，Ansible 任务运行的过程中需要用户输入一些数据，这些数据要么比较

私密不方便保存，或者数据是动态的，不同用户有不同的需求，比如输入用户自己的账号和密码或者输入不同的版本号会触发不同的后续操作等。Ansible 的 `vars_prompt` 关键字就是用来处理上述这种与用户交互的情况的。

我们先来看一个例子：我们需要用户提供自己的账号和密码来登录自己的网络账户，并且可以给用户以适当的文字提示。代码如下所示：

```
---
- hosts: all
  vars_prompt:
    - name: share_user
      prompt: "What is your network username?"
    - name: share_pass
      prompt: "What is your network password?"
      private: yes
```

 **提示** 为了安全起见，命令行上面输入的任何字符默认都是不可见的。

关键字 `vars_prompt` 下面几个常用的选项总结如下。

- ❑ `private`：该值为 `yes`，即用户所有的输入在命令中默认都是不可见的；而将其值设为 `no` 时，用户输入可见。
- ❑ `default`：为变量设置默认值，以节省用户输入时间。
- ❑ `confirm`：特别适合输入密码的情况，如果将值设为 `yes`，则会要求用户输入两次，以增加输入的正确性。

“输入提示”这种交互式的操作方法可以很方便地让用户输入特定的信息，但是在大部分情况下我们应尽量避免使用这一功能，除非是在非常必要时，因为交互操作虽然满足了用户的个性化需求，但是大大降低了 Ansible 自动化运维的能力。

5.7 Tags 标签

默认情况下，Ansible 在执行一个 Playbook 时，会执行 Playbook 中定义的所有任务。Ansible 的标签（Tags）功能可以给角色（Roles）、文件、单独的任务甚至整个 Playbook 打上标签，然后利用这些标签来指定要运行 Playbook 中的个别任务，或不执行指定的任务，并且它的语法非常简单。

在下面这个例子中，我们将展示多种不同的打标签的方法。

```
---
# 可以给整个 Playbook 的所有任务打一个标签
- hosts: webservers
  tags: deploy
```

```

roles:
  # 给角色打的标签将会应用于角色下所有的任务
  - { role: tomcat, tags: ['tomcat', 'app'] }

tasks:
  - name: Notify on completion.
    local_action:
      module: osx_say
      msg: "{{inventory_hostname}} is finished!"
      voice: Zarvox
    tags:
      - notifications
      - say

  - include: foo.yml
    tags: foo

```

假设我们将上述代码保存在文件 `tags.yml` 中，我们可以通过执行下面这条命令来只执行“Notify on completion.”任务。

```
ansible-playbook tags.yml --tags " say"
```

如果我们想跳过带有“notifications”标签的任务，可以使用 `--skip-tags` 选项。

```
ansible-playbook tags.yml --skip-tags "notifications"
```

正如我们所看到的，只要我们在 Playbook 中打好完整的标签，我们就可以非常方便地对 Playbook 中的众多任务进行抽取，有针对性地执行任务，或者跳过那些暂时不需要执行的任务。

我们可以为一个对象添加多个标签，但是在添加多标签时必须使用 YAML 列表格式。YAML 列表格式如下：

```

# 最简洁的写法
tags: ['one', 'two', 'three']

# 最清晰的写法
tags:
  - one
  - two
  - three

# 不正确的写法
tags: one, two, three

```

通常情况下，笔者只在内容比较多的 Playbook 中使用标签功能，尤其是在包含了独立角色或任务的 Playbook 中。在一些结构比较简单且内容量比较小的 Playbook 中，并不建议大量使用标签功能，因为这会在一定程度上增加视觉混乱。当然这还要根据个人爱好而定，找到适合自己的标签风格，适合自己的才是最好的。

5.8 Block 块

Ansible 从 2.0.0 版本开始引入了块功能，块功能可以将任务进行分组，并且可以在块级别上应用任务变量。同时，块功能还可以使用类似于其他编程语言处理异常那样的方法，来处理块内部的任务异常。

我们来看下面一个例子。

```
---
- hosts: web
  tasks:
    # Install and configure Apache on RedHat/CentOS hosts.
    - block:
      - yum: name=httpd state=present
      - template: src=httpd.conf.j2 dest=/etc/httpd/conf/httpd.conf
      - service: name=httpd state=started enabled=yes
    when: ansible_os_family == 'RedHat'
    sudo: yes

    # Install and configure Apache on Debian/Ubuntu hosts.
    - block:
      - apt: name=apache2 state=present
      - template: src=httpd.conf.j2 dest=/etc/apache2/apache2.conf
      - service: name=apache2 state=started enabled=yes
    when: ansible_os_family == 'Debian'
    sudo: yes
```

在上例中，我们使用了带有 `when` 语句的块来指定在不同平台上运行一组不同的安装配置任务，我们可以看到，块将 `apt`、`template`、`service` 三个模块任务包含在内作为一个整块，这样就不用每一个模块任务后都跟一个 `when` 语句进行操作系统的判断了。由此我们可以看出，块功能非常适合于多个任务共用同一套任务参数的情况。

块功能也可以用来处理任务的异常。比如有一个 Ansible 任务是监控一个并不太重要的应用，这个应用的正常运行与否对后续的任务并不产生影响，这时我们就可以通过块功能来处理这个应用的报错。如下代码所示：

```
tasks:
  - block:
    - name: Shell script to connect the app to a monitoring service.
      script: monitoring-connect.sh
    rescue:
      - name: 只有脚本报错时才执行
        debug: msg="There was an error in the block."
    always:
      - name: 无论结果如何都执行
        debug: msg="This always executes."
```

当块中的任意任务出错时，`rescue` 关键字对应的代码块就会被执行，而 `always` 关键字对

应的代码块无论如何都会被执行。

如果你需要创建一个健壮可靠的 Playbook，那么块功能将是一个非常不错的选择。但是，就像其他所有语言中的异常处理语句一样，块功能的异常处理也会在一定程度上将代码复杂化。根据自己的实际情况，如果具体项目中可以很方便地在每个任务后面使用 `failed_when` 语句来对异常进行处理，并且可以维持幂等性，或者可以以别的方式重构你的 Playbook，那么就没有必要使用块功能来处理任务异常。

5.9 本章小结

Playbook 是 Ansible 实现自动化管理的最主要方式。通过对本章内容的学习，应该了解到如何使用变量、Inventory 文件、Handlers、条件判断、标签等诸多功能。对这些 Ansible 基础组件的了解越多，理解越深刻，那么将来在使用 Ansible 创建或扩展自己的自动化运维架构的时候，就会更加高效，更加游刃有余。

includes 使用场景中的案例。

该场景中一共有 A、B、C、D、E、F 这 6 个 Project (项目), 但均需要用到 Restart PHP Process (重启 PHP 过程) 功能。此时, 我们可以把 Restart PHP Process 功能作为一个 playbook 文件发

布给 A、B、C、D、E、F 这 6 个项目, 而不必每个项目都单独写一个 Restart PHP Process 功

能, 这也就是 includes 使用场景。

如何来实现, 我们稍后介绍。

第二篇 Part 2

高级进阶篇

如注释所示, 该 YML 文件完成重启功能。

前面章节我们介绍了一个 playbook 文件, 但在实际应用中, 我们经常会遇到需要重复执行某些任务的情况。这时, 我们可以使用 includes 功能, 将一个 playbook 文件包含到另一个 playbook 文件中。这不仅可以避免重复编写代码, 还可以提高 playbook 文件的可读性和维护性。本章将详细介绍 includes 的使用方法, 包括如何定义 includes 任务、如何调用 includes 任务以及 includes 任务的优缺点。通过本章的学习, 读者将能够熟练地使用 includes 功能, 从而提高 playbook 文件的编写效率和可维护性。

- 第 6 章 Playbook 高级技巧进阶
- 第 7 章 Inventory 文件扩展
- 第 8 章 Ansible 插件扩展
- 第 9 章 Ansible 企业应用实战
- 第 10 章 Ansible 基于 Windows 的管理架构
- 第 11 章 Ansible 安全优化篇

.....

应的代码块，在何时都会被执行。

如果你使用 `failed_when` 的 Playbook，那么该功能将是一个非常不错的选择。但是，就像其他异常处理语句一样，该功能的异常处理也会在一定程度上将代码复杂化。在实际情况，如果具体项目中可以很方便地在每个任务后面使用 `failed_when` 来处理，并且可以维持幂等性，或者可以针对异常任务进行重试，那么使用该功能来处理任务异常。

Chapter 6 第6章

Playbook 高级技巧进阶

5.9 本章小结

Playbook 是 Ansible 实现自动化管理的最主要方式。通过对本章内容的学习，了解到如何使用变量、Inventory 文件、Handlers、Modules 等功能。对这些 Ansible 基础组件的了解越多，理解越深刻，那么将来在使用 Ansible 创建或扩展自己的自动化结构的时候，就会更加高效，更加游刃有余。

前面所有章节介绍的案例均写为一个 Playbook 文件，但在实际工作中，一个完整的项目往往是很多功能的组合体，如果将所有的功能写在一个 Playbook 中会存在问题，如：代码耦合程度高；Playbook 过长而维护成本巨大；Playbook 过于臃肿而缺乏灵活性等。即使我们将一个 Playbook 分割成多个小的 Playbook 文件，但也无法从根本上解决如上问题。这时我们更需要考虑的是我们使用方式是否存在问题。为解决如上问题，本章为大家介绍 Ansible Playbook 高级语法技巧。接下来的内容，大家会接触到 Includes、Handlers、Files、Templates、Roles、Jinja、Galaxy 等新名词，同时也会用 Roles 改造已有的 Includes 代码，使用 Templates 结合 Jinja 生成配置模板等。

6.1 巧用 Includes

Includes 在 Ansible 中主要起引用功能，其功能非常强大，不仅可以引用 Playbook 的 YML 文件，而且 Vars、Handlers、Files 也支持 Includes 的引用。

6.1.1 Includes 使用场景

有时，我们发现大量的 Playbook 内容需要重复编写，各 Tasks 之间功能需相互调用才能完成各自功能，Playbook 庞大到维护困难，这时我们需要使用 Includes。如图 6-1 所示为

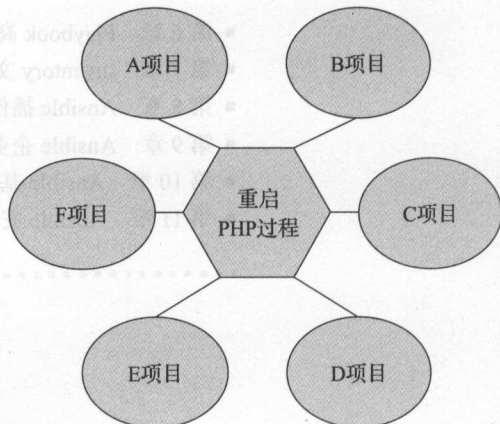


图 6-1 Includes 使用场景

Includes 使用场景中的案例。

该场景中共有 A、B、C、D、E、F 这 6 个 Project (项目)，但均需使用 Restart PHP Process (重启 PHP 过程) 功能，此时，我们可以把 Restart PHP Process 功能作为单独的 Playbook 文件独立出来，以方便其他项目 Includes (引用)，而不必每个项目都重新写 Restart PHP Process 功能，这种场景下就需要使用 Includes。

我们一起看下具体如何实现。

1) RestartPHPProcess.yml 的配置如下：

```
---
- hosts: phpserver                                # 命令执行对象为 phpserver 的主机 (组)
  remote_user: root                                # 远程主机执行用户为 root

  tasks:
    - name: RestartPHPProcess                      # 该 Task 名为 RestartPHPProcess
      service: name=php-fpm state=restarted        # 调用 service 模块，重启名为 php-
                                                    # fpm 的服务
```

如注释所示，该 YML 文件完成重启 php-fpm 功能。

2) A Project 的配置如下：

```
---
# This playbook deploys the whole application stack in this site.

- hosts: localhost
  remote_user: root

  tasks:
    - name: A Project command
      command: A Project command

    - name: RestartPHPProcess
      hosts: phpserver
      remote_user: root

      tasks:
        - include: RestartPHPProcess.yml          # 引用 RestartPHPProcess.yml 文件，
                                                    # 调用该文件中定义的功能集
```

该 YML 文件除完成 A Project 自身功能外，还通过 Include: RestartPHPProcess.yml 调用该 YML 定义的功能，同时完成 php-fpm 的重启工作。

B、C、D、E、F Projects 同样以如上 Include 的方式来重启 php-fpm 即可，而不必每次都重写该功能。下一节会带大家逐步深入了解 Includes 的使用。

6.1.2 Includes 用法

了解了 Includes 的使用场景后，我们继续了解 Includes 的具体用法。本节我们使用的案

例是：Ansible 结合 Git^①完成指定版本的拉取及项目目录初始化。在 6.2 节我们会使用 Roles 重构该案例，为大家演示如何优化代码框架。Includes 在 Playbook 中使用方式很简单，格式也非常简洁，请参考如下的代码：

```
tasks:
- include: included-playbook.yml
```

其中，included-playbook.yml 的内容如下：

```
---
- name: Add profile info for user.
  copy:
    src: example_profile
    dest: "/home/{{ username }}/.profile"      # copy 功能模块
    owner: "{{ username }}"                    # copy 模块 owner 属性
    group: "{{ username }}"                    # copy 模块 group 属性
    mode: 0744                                 # copy 模块 mode 属性

- name: Add private keys for user.
  copy:
    src: "{{ item.src }}"
    dest: "/home/.ssh/{{ item.dest }}"
    owner: "{{ username }}"
    group: "{{ username }}"
    mode: 0600
  with_items: ssh_private_keys

- name: Restart example service.
  service: name=example state=restarted      # service 模块，重启名为 example 的服务

这些任务负责检出 git 变量
- name: create dir
  # file 模块，用于设置文件属性
  file: path="{{ package_dir }}{{ project }}"-release-{{ git_commit }}/{{ flag }}
        {{ project }} owner=www group=www mode=0755 recurse=yes state=directory

# 这些任务负责检出 git 变量

- name: Git pull
  git: repo={{ repository_static }} dest={{ package_dir }}{{ project }}"-release-{{
    git_commit }}/{{ flag }}-{{ project }} version="{{ git_commit }}" force=yes #
    git 模块，结合 Ansible 后用于自动化版本管理
  # 这些任务负责检出 git 变量

- name: Git init before git pull
  command: /usr/bin/git fetch
  args:
    chdir: "{{ package_dir }}{{ project }}"-release-{{ git_commit }}/{{ flag }}-{{
```

① 分布式版本控制软件，官方地址：<https://git-scm.com/>。

```

project }}"

- name: Git reset
  command: /usr/bin/git reset --hard
  args:
    chdir: "{{ package_dir }}{{ project }}-release-{{ git_commit }}/{{ flag }}-{{
    project }}"

- name: Git checkout
  command: /usr/bin/git checkout {{ git_commit }}
  args:
    chdir: "{{ package_dir }}{{ project }}-release-{{ git_commit }}/{{ flag }}-{{
    project }}"

```

在这个案例中，included-playbook.yml 做了下面几件事情：

- 1) 初始化用户环境变量；
- 2) 添加 key 认证；
- 3) 重启服务；
- 4) 递归初始化 git 文件存放目录并设置目录属主属组为 www.www；
- 5) Git pull 指定版本的 git 库至指定目录；
- 6) Git checkout 最新指定版本至指定目录。

该案例完成这样一件任务：新安装系统的服务器，初始化程序用户并添加 key 认证，结束后分发程序软件包至指定目录。这其中使用了很多变量，“{{ }}" 中的内容均为变量。大家也可以看到 included-playbook.yml 中的内容非常多，这将提高代码的维护难度，同时该配置中每个功能块都可以独立成一个模块，所以朋友们可以利用 Includes 再次分割。我们再做一些优化，将每个可复用的功能均模块化。我们将其分别拆分为如下几个文件。

- 1) user-config.yml：完成用户初始化工作。

```

---
- name: Add profile info for user.
  copy:
    src: example_profile
    dest: "/home/{{ username }}/.profile"
    owner: "{{ username }}"
    group: "{{ username }}"
    mode: 0744

- name: Add private keys for user.
  copy:
    src: "{{ item.src }}"
    dest: "/home/.ssh/{{ item.dest }}"
    owner: "{{ username }}"
    group: "{{ username }}"
    mode: 0600

```

```

with_items: ssh_private_keys
- name: Restart example service.
  service: name=example state=restarted

```

2) create_dir.yml: 完成目录初始化工作。

```

---
# 这些任务负责检出 git 变量
#
- name: create dir
  file: path={{ package_dir }}{{ project }}-release-{{ git_commit }}/{{ flag }}-{{
    project }} owner=www group=www mode=0755 recurse=yes state=directory

```

3) static_git_pull.yml: 完成拉取 git 代码工作。

```

---
# 这些任务负责检出 git 变量
#
- name: Git pull
  git: repo={{ repository_static }} dest={{ package_dir }}{{ project }}-release-{{
    git_commit }}/{{ flag }}-{{ project }} version="{{ git_commit }}" force=yes

```

4) git_checkout.yml: 完成 git 初始化及指定版本拉取工作。

```

---
# 这些任务负责检出 git 变量
#
- name: Git init before git pull
  command: /usr/bin/git fetch
  args:
    chdir: "{{ package_dir }}{{ project }}-release-{{ git_commit }}/{{ flag }}-{{
      project }}"
- name: Git reset
  command: /usr/bin/git reset --hard
  args:
    chdir: "{{ package_dir }}{{ project }}-release-{{ git_commit }}/{{ flag }}-{{
      project }}"
- name: Git checkout
  command: /usr/bin/git checkout {{ git_commit }}
  args:
    chdir: "{{ package_dir }}{{ project }}-release-{{ git_commit }}/{{ flag }}-{{
      project }}"

```

将如上模块通过 Includes 模块再次封装，我们重命名新的 Playbook 名为 Sysinit.yml。

5) Sysinit.yml: 完成所有任务的调度和执行工作。


```

---
- include: user-config.yml
  vars:
    username: johndoe
    ssh_private_keys:
      - { src: /path/to/johndoe/key1, dest: id_rsa }
      - { src: /path/to/johndoe/key2, dest: id_rsa_2 }
      - include: user-config.yml

- include: janedoe.yml
  vars:
    username: janedoe
    ssh_private_keys:
      - { src: /path/to/janedoe/key1, dest: id_rsa }
      - { src: /path/to/janedoe/key2, dest: id_rsa_2 }

- include: ../create_dir.yml
- include: ../static_git_pull.yml
- include: ../git_checkout.yml

```

通过如上 Includes 引用方式的好处不言而喻：简洁，干净，解耦，复用度高，易于维护。所以掌握好 Includes 很重要。

6.1.3 动态 Includes

我们已经看到 Includes 功能的强大，不仅如此，Ansible 还提供了动态加载 Includes 功能，即在满足一定条件时加载 Includes。这个功能极大地提高了 Ansible Includes 功能的灵活性和可扩展性。我们通过下面的案例进一步了解。

```

# 引用附加的任务，该任务只在运行时有效
- name: Check if extra_tasks.yml is present.
  stat: path=extras/extra-tasks.yml      # 判断 extras 目录下 extra-tasks.yml 文件
                                          # 是否存在
  register: extra_tasks_file            # 获取状态返回值
  connection: local

- include: tasks/extra-tasks.yml        # 结合如下 when 条件，只有当 extra_tasks_file
                                          # 文件存在时再加载 include

  when: extra_tasks_file.stat.exists

```

通过使用 when 条件判断，Ansible 加强了 Includes 对不确定因素的处理机制，在健壮程序代码的环节中扮演着重要角色，强烈建议熟练掌握。

6.1.4 Handler Includes 使用技巧

Ansible Hanlder 结合 Notify^①主要用于当资源状态发生变化时一次性地执行指定操作。Handlers 也支持 Includes 功能，用法和 Tasks 的调用方式一样，只是要写在 Hanlers 区域。举个简单的例子。

① Ansible 内置功能之一，结合 Handlers 实现当状态变化后一次性地执行指定操作。

roles/logsync/handlers/ 的目录结构如下：

```
roles/logsync/handlers/
├── epel.yml
├── main.yml
└── syncinstall.yml
```

我们希望 main.yml 引用 epel.yml 和 syncinstall.yml，通过如下方式即可实现：

```
---
```

```
- include: epel.yml # 引用 epel.yml 文件
- include: syncinstall.yml # 引用 syncinstall.yml 文件
```

通过如上方式，我们即实现了 Handler 文件的 Includes 引用。

6.1.5 Playbooks Includes 使用技巧

Ansible 同样允许 Playbook Include Playbook，使用方式与 Task、Handlers 一样，下面示例可供参考。

```
- hosts: all
  remote_user: root

  tasks:
    [...]

    - include: web.yml # 引用 web.yml playbook
    - include: db.yml # 引用 db.yml playbook
```

通过如上方式，我们可以创建一个主 Playbook 文件，通过 Includes 加载其他独立的 Playbook，当我们需要执行全部命令时，只要通过一条命令执行主 Playbook 文件即可，如希望针对某功能变更，只执行对应的 Playbook 文件即可。

6.2 巧用 Roles

通过 6.1 节的学习可知，Includes 使 Playbook 更加整洁有效，清晰健壮。但放眼我们日常工作，一个项目从开始至结束不是简单数个或数十个 Playbook 就可以完成所有功能，如笔者曾经完成的自动化发布项目，总文件数达 150 个，如果仅通过简单的 Includes 不停地引用，那最终的结果难以想象，恐怕只能用俄罗斯套娃（是俄罗斯特产木制玩具，一般由多个一样图案的空心木娃娃一个套一个组成，一般在 6 个以上）来形容了。

除套用复杂外，一些实际额外业务场景也需考虑在内，如：变更指定主机或主机组；命名不规范，维护和传承成本大；某些功能集需 Includes 多个 Playbook 方可实现等。这些场景在企业中常存在，尤其 IT 行业高流动性的特征使这些问题更显突出。有没有办法解决这些

问题呢？Ansible Roles 不仅可以解决这些问题，而且可以做到更多。

Roles 是 Ansible 1.2 版本新加入的功能，字面意思是角色，大家可以理解为：有相互关联功能的集合。相对 Includes 功能，Roles 更适合于大项目 Playbook 的编排架构。简而言之，Ad-Hoc 适用于临时命令的执行，Playbook 合适中小项目，而大项目一定使用 Roles。Roles 不仅支持 Tasks 的集合，同时包括 vars_files、tasks、handlers、meta、templates。

6.2.1 构建 Roles

Roles 主要依赖于目录的命名和摆放，默认 tasks/main.yml 是所有任务的入口，所以使用 Roles 的过程可以理解为目录规范化命名的过程。如下两个目录就可以构建 Ansible Roles。

```
role_name/
meta/
tasks/
```

每个目录下均由 main.yml 定义该功能的任务集，tasks/main.yml 默认执行所有指定的任务。Roles 的调用文件 playbook_role.yml 的内容如下：

```
---
- hosts: all

roles:
  - role_name
```

Roles 执行方法如下：

```
ansible-playbook playbook_role.yml
```

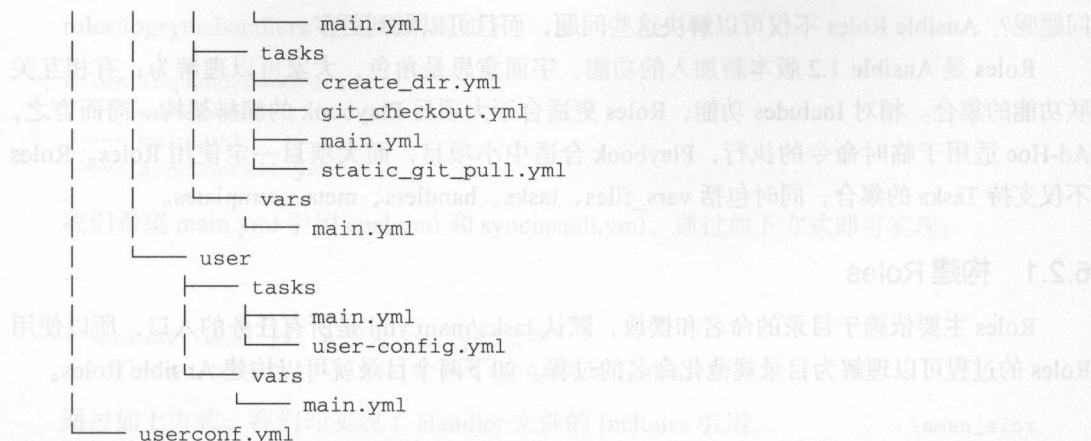
Roles 目录可以摆放在 /etc/ansible/ansible.cfg 中 “roles_path” 定义的路径，也可以和入口 Playbook 文件存放在同级目录，Ansible 对此没有强制要求。但建议将代码存放在代码机预先规划的目录，以便管理。

6.2.2 使用 Roles 重构 Playbooks

Roles 在企业复杂业务场景中应用频率最高，适用场景最多。本节我们通过重构 6.1 节代码，为大家展示 Roles 用法。

Roles 严重依赖目录命名规则和目录摆放，所以目录的命名和目录摆放非常重要。6.1 节代码案例使用 Roles 重构后目录结构如下（使用 tree 命令返回的结果）：

```
fab2ansible
├── group_vars
│   └── all
├── roles
│   ├── git
│   └── files
```



没有代码基础的朋友如果觉得只通过代码形容不大清楚的话，我们专门整理了 Roles 各模块调用关系，具体 Roles 模块调用结构如图 6-2 所示。

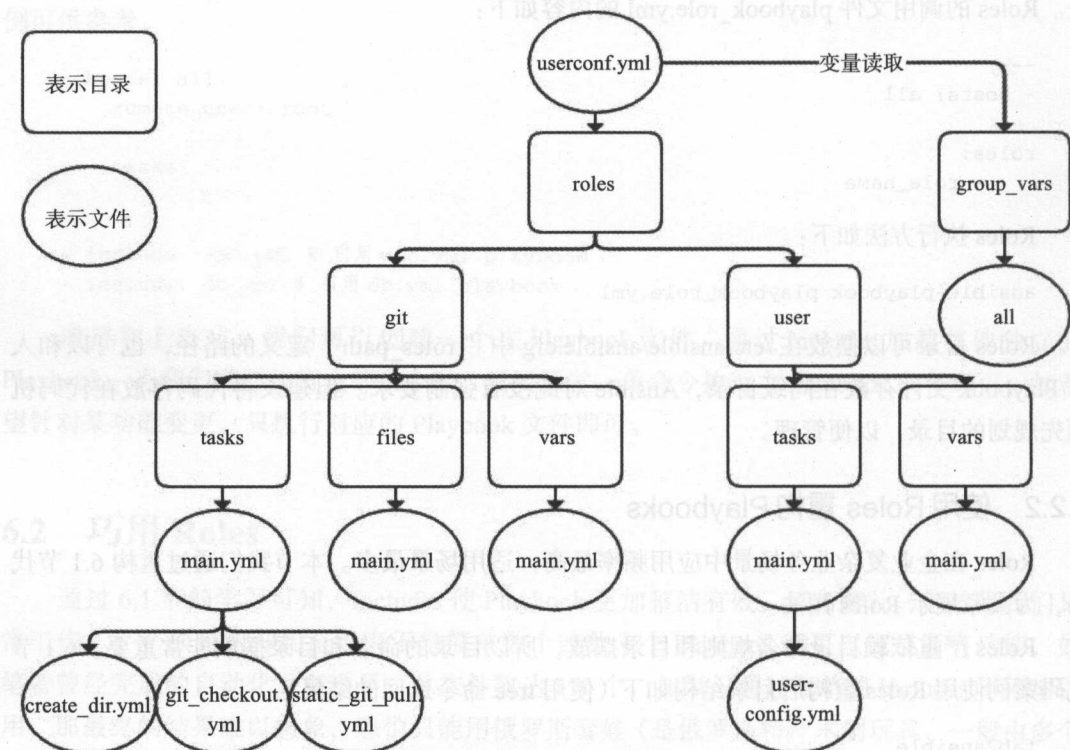


图 6-2 Roles 模块调用结构

如我们希望调用 roles 目录中的 git 模块，参考图 6-2 目录结构我们依次自上而下剖析代码。

1) group_vars/all 文件：定义 roles 变量，编辑内容如下。

以下列出的变量对所有主机和组有效，即全局有效

```
flag: king
javaflag: java
tech: php
damn: '-'
git: git
svn: svn
package_dir: /srv/depoy/
project_dir: /srv/www/
project: cp
```

group_vars 目录下的文件定义 Roles 中调用的变量，Roles 对应调用该目录同名文件中定义的变量，文件名为 all 的文件定义的变量针对所有 Roles 生效。

2) userconf.yml 文件：设置调用 Roles 的 git 模块，编辑内容如下。

该 playbook 用于初始化用户配置

```
- hosts: localhost
  remote_user: root
```

```
  roles:
    - role: git
```

roles 为关键字，role:git 表示调用 roles 的 git 模块。如希望同时调用图 6-2 中的 user 模块，于该行下同级别对齐添加如下配置即可。

```
- role: user
```

如需继续添加功能，方式一样。

3) roles/git/tasks/main.yml 文件：设置调用 git 模块实现的功能，编辑内容如下。

```
---
- include: create_dir.yml
- include: static_git_pull.yml
- include: git_checkout.yml
```

通过 include 引用 create_dir.yml、git_pull.yml、git_checkout.yml、git_pull.yml 功能模块，如希望增加其他功能模块，于该行下同级别对齐添加如下配置即可。

```
- include: git_push.yml
```

4) create_dir.yml、git_checkout.yml、static_git_pull.yml 文件：设置我们希望完成的具体功能。具体代码如下，部分代码会做解释。

create_dir.yml 代码如下:

```
---
# 这些任务负责检出 git 变量
#

- name: create dir
  file: path={{ package_dir }}{{ project }}-release-{{ git_commit }}/{{ flag }}-{{ project }} owner=www group=www mode=0755 recurse=yes state=directory
```

{{ }} 在 Ansible 中表示变量引用, create_dir.yml 要实现的功能是递归创建目录并定义目录属主、属组权限为 www 用户。

static_git_pull.yml 的作用是拉取指定的 git 版本至指定目录。代码如下:

```
---
# 这些任务负责检出 git 变量
#

- name: Git pull
  git: repo={{ repository_static }} dest={{ package_dir }}{{ project }}-release-{{ git_commit }}/{{ flag }}-{{ project }} version="{{ git_commit }}" force=yes
```

git_checkout.yml 实现 Git 项目初始化, Checkout 最新代码至指定目录。代码如下:

```
---
# 这些任务负责检出 git 变量
#

- name: Git init before git pull
  command: /usr/bin/git fetch
  args:
    chdir: "{{ package_dir }}{{ project }}-release-{{ git_commit }}/{{ flag }}-{{ project }}"

- name: Git reset
  command: /usr/bin/git reset --hard
  args:
    chdir: "{{ package_dir }}{{ project }}-release-{{ git_commit }}/{{ flag }}-{{ project }}"

- name: Git checkout
  command: /usr/bin/git checkout {{ git_commit }}
  args:
    chdir: "{{ package_dir }}{{ project }}-release-{{ git_commit }}/{{ flag }}-{{ project }}"
```

代码中 chdir: 下的 “{{ package_dir }}” 需要用双引号引起来, 在 Ansible 中变量引用有些地方需要使用双引用号而有些地方不需要, 这因为 Ansible 的源码是多人协作方式编写的, 所以变量引用部分建议大家多关注官网最新变化。git_checkout.yml 的功能很简单, 都是调用

git 命令，git 初始化后，checkout 最新代码至指定目录。

至此，6.1 节代码改造完毕，较 Includes 而言，Roles 将功能集以文件夹方式模块化后，使代码更为整洁，更为灵活，复用性也更高，变量的调用方式、Roles 的命名方式等使其在架构上更为规范化，在复杂度较高的项目中体现得更加明显。

6.2.3 Roles 技巧之 Handlers：动态变更

如 6.2.1 节介绍，Roles 不仅支持 Tasks 调用，同时支持 vars、files、handlers、meta、templates 的调用。本节为大家介绍 Handlers 在 Roles 中的使用技巧。

客观讲，“动态变更”形容不是很恰当，官方原文为“Running Operations On Change”，即仅当状态变化时才会执行命令。Handlers 通常和 Notify 搭配使用，当（文件、进程、返回等）状态有变化时，Notify 会通过 Handlers 做指定的变更。我们通过一个功能完整的 Roles 来整体了解 Vars、Files、Handlers、Meta、Templates，然后逐步深入 Roles Handlers 用法。请看示例 example.yml：

```
site.yml
webservers.yml
fooservers.yml
roles/
```

```
  common/
    files/
    templates/
    tasks/
    handlers/
    vars/
    defaults/
    meta/
  webservers/
    files/
    templates/
    tasks/
    handlers/
    vars/
    defaults/
    meta/
```

example.yml 为大家展示了两个功能齐全的 Roles，分别为 common 和 webservers，每个 Roles 均包括 files、templates、tasks、handlers、vars、defaults、meta。在 Playbooks 中的调用方式如下：

```
---
- hosts: webservers
  roles:
    - common
    - webservers
```

了解了 Roles 支持的功能集和调用方式后，我们再来了解这些功能集的含义。

- ❑ roles/x/tasks/main.yml: 主函数，包括在其中的所有任务将被执行。
- ❑ roles/x/handlers/main.yml: 所有包括其中的 handlers 将被执行。
- ❑ roles/x/vars/main.yml: 所有包括在其中的变量将在 roles 中生效。
- ❑ roles/x/meta/main.yml: roles 所有依赖将被正常登入。
- ❑ roles/x/{files, templates, tasks}/(dir depends on task): 所有文件、模板都可存放在这里，放在这里最大的好处是不用指定绝对路径。

接下来，我们通过如下案例来学习 Handlers 的应用场景。

案例场景：当 Apache 的配置文件发生变化时重启 Apache 进程。

步骤 1：编排 Roles 目录结构如下。

```
roles/apache/
├── handlers
│   └── main.yml
└── tasks
    ├── restart.yml
    └── main.yml
```

目录结构需按要求编排，不得随意变更名称。

步骤 2：编辑 roles/apache/handlers/main.yml 的内容如下。

```
---
# sleep 10s
#
- name: restart apache
  Service: name=apache state=restarted
```

该 YML 要实现的功能非常简单：重启 Apache 进程。

步骤 3：编辑 roles/apache/tasks/restart.yml 内容如下。

```
---
- name: transfer apache config
  copy: src=httpd.conf dest=/opt/apache/httpd.conf
  notify:
    - restart apache
```

该 YML 功能为更新 Apache 配置文件，如配置文件有变化则重启 Apache。

步骤 4：编辑 roles/apache/tasks/main.yml 内容如下。

```
---
- include: restart.yml
```

步骤 5：编辑 Roles 同级目录 apache.yml 文件，内容如下。

```
---
```



```

- hosts: webserver
  remote_user: root

  roles:
    - role: apache

```

该 YML 为总调度文件，完成 Apache 配置文件的变更和 Apache 的重启工作。

步骤 6: 执行命令 `ansible-playbook apache.yml`，验证结果。

命令运行结果为更新 Apache 配置文件，如配置文件有更新则重启 Apache，如无错误返回为正常。

如上，我们即完成了当 Apache 配置文件有更新时才重启 Apache。在实际工作中我们并不希望随意重启应用造成服务中断，所以 Handlers 非常有用。

6.2.4 Roles 技巧之 Files：文件传输

Files 和 Templates 均用于 Ansible 文件处理，两者主要区别是：Files（不是 file 模块）目录下的文件无需写绝对路径即可将文件传输至远程主机；Templates 目录下的文件以 Jinja2 渲染，且传输文件至远程主机的同时支持预定义变量替换。接下来我们看 Roles 中 Files 的使用方式。

案例场景：

将 example role 下的 MAGEDU.PPT 和 STANLEY.PPT 两个文件传输至远程，并修改文件名均为英文小写。

步骤 1: 编排目录结构如下。

```

file.yml
roles/example/
|   |___ files
|   |   |___ MAGEDU.PPT
|   |   |___ STANLEY.PPT
|   |___ tasks
|   |   |___ file.yml
|   |   |___ main.yml

```

步骤 2: 依次创建文件 MAGEDU.PPT、STANLEY.PPT。

MAGEDU.PPT 内容如下：

This is magedu.ppt file.

STANLEY.PPT 内容如下：

This is stanley.ppt file.

步骤 3: 依次编辑 `./file.yml`、`./roles/example/tasks/file.yml`、`./roles/example/tasks/main.yml`。

`./file.yml` 内容如下：

```
---
# 该 playbook 是整个项目的调度入口
```

```
- hosts: 192.168.37.142
  remote_user: root
  gather_facts: false

  roles:
    - role: example
```

./roles/example/tasks/file.yml 内容如下:

```
---
- name: file change example
  # copy: src=MAGEDU.PPT dest=/data/magedu.ppt owner=stanley group=stanley
  copy: src={{ item.src }} dest=/data/{{ item.dest }} owner=stanley group=stanley
  with_items:
    - { src: 'MAGEDU.PPT', dest: 'magedu.ppt' }
    - { src: 'STANLEY.PPT', dest: 'stanley.ppt' }
```

./roles/example/tasks/main.yml 内容如下:

```
---
- include: file.yml
```

步骤 4: 传输文件到远程主机并修改文件名为英文小写。

在 roles 目录同级目录下执行命令:

```
ansible-playbook file.yml
```

返回结果如下:

```
PLAY [192.168.37.142] *****
TASK: [example | file change example] *****
changed: [192.168.37.142] => (item={'dest': 'magedu.ppt', 'src': 'MAGEDU.PPT'})
changed: [192.168.37.142] => (item={'dest': 'stanley.ppt', 'src': 'STANLEY.PPT'})

PLAY RECAP *****
192.168.37.142 : ok=1    changed=1    unreachable=0    failed=0
```

没有 False 返回即为正常。我们回头再来整理下本次的执行逻辑, 如图 6-3 所示。

Ansible-playbook 调用执行 file.yml (和 roles 目录同级的 file.yml 文件), 随后根据文件内容依次调用 ./roles/example/tasks/main.yml、./roles/example/tasks/main.yml、./roles/example/files/{MAGEDU.PPT, STANLEY.PPT}。此过程严重依赖目录结构及命名, 请勿随意更改目录及文件名。

Roles 的 Files 功能设计主要针对业务文件传输需求, 凡存放于对应 Roles 的 Files 目录下

的文件，传输时只需指定相对路径即可，这在很大程度上保证了管理机故障迁移时 Ansible 的健壮性，同时也从规则上使使用者有意规范自己的文件存放习惯。在企业中我们不仅会遇到文件传输的需求，对于应用的配置文件，针对不同的主机需要进行相应的配置变更该怎么办呢？Templates 可以满足我们需求。

6.2.5 Roles 技巧之 Templates：模板替换

Templates 常被用作传输文件，同时支持预定义变量替换。因 Templates 由 Jinja2 渲染格式，所以介绍 Templates 前我们先来了解 Jinja2。

Jinja2 是什么？Jinja2 是一个 Python 的功能齐全的模板引擎。它有完整的 Unicode 支持，一个可选的集成沙箱执行环境，被广泛使用，以 BSD 许可证授权。

本次内容我们只需掌握 Jinja2 读取变量的方式即可，关于 Jinja2 更详细内容，我们在 6.3 节详细介绍。通常两个花括号中间加变量名且花括号和变量名间需有空格分隔，如 `{{ variable }}`，更详尽内容请参考 Jinja2 官网 <http://jinja.pocoo.org/>。我们结合如下案例来进一步了解 Tempaltes 的使用。

案例场景：将 `order.j2` 分发至远程主机 `/data/{{ PROJECT }}` 目录下，并改名为 `order.conf`，且替换配置文件中变量为对应的值。接下来我们逐步实现该功能。

步骤 1：编排目录如下。

```
template.yml
roles/template/
|  tasks
| |  main.yml
| |  template.yml
|  templates
| |  order.j2
|  vars
|  main.yml
```

该目录结构下，`order.j2` 模板文件必须存放于 `templates` 目录下，`main.yml` 变量文件存放于 `vars` 目录下。`template.yml` 和 `main.yml` 任务执行文件存放于 `tasks` 目录下。目录结构及命名方式不得随意变更。

步骤 2：依次编辑 `tempates.yml`（和 `roles` 目录同级）任务总调度文件。

```
---
# 该 playbook 是整个项目的调度入口
```

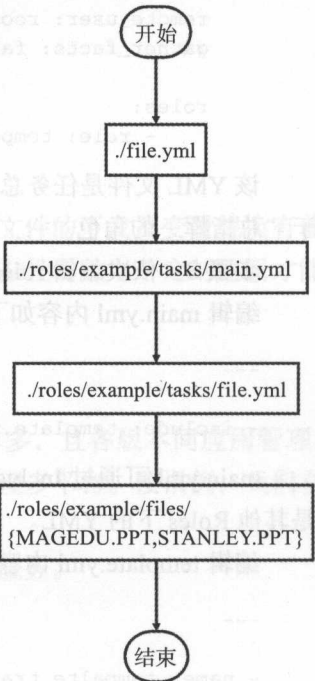


图 6-3 执行逻辑

```

- hosts: 192.168.37.142
  remote_user: root
  gather_facts: false

  roles:
    - role: template

```

该 YAML 文件是任务总调用文件，主要指定远程主机、执行用户、调用的 roles 等，相当于“总指挥”的角色。

步骤 3：依次编辑 roles/template/tasks/{main.yml, template.yml} 任务定义文件。

编辑 main.yml 内容如下：

```

---
- include: template.yml

```

main.yml 可通过 Include 灵活引用所需的功能组件，不仅是当前目录下的 YAML，也可以是其他 Roles 下的 YAML。

编辑 template.yml 内容如下：

```

---
- name: template transfer example
  template: src=order.j2 dest=/data/{{ PROJECT }}/order.conf

```

{{ PROJECT }} 的变量引用方式即本节伊始提到的 Jinja2 格式。源文件是 order.j2，远程目录及目的文件名分别是 /data/{{ PROJECT }}/ 和 order.conf。

步骤 4：编辑 roles/template/templates/order.j2，定义模板文件。

```

project: {{ PROJECT }}
switch: {{ SWITCH }}
dbport: {{ DBPORT }}

```

步骤 5：编辑 roles/template/vars/main.yml，定义变量。

```

---
PROJECT: "JAVA"
SWITCH: "ON"
DBPORT: "3306"

```

变量的定义方式和我们日常 Shell 脚本变量定义差别不大，6.3.7 我们会引申 Jinja2 更多的变量定义方式。

步骤 6：我们来执行命令并看下返回及结果。

执行命令：

```
ansible-playbook template.yml
```


登录远程主机 192.168.37.142，检查文件是否正常下发，以及模板中的变量是否被正常替换。

```
cat /data/JAVA/order.conf
project: JAVA
switch: ON
dbport: 3306
```

order.conf 内容如上表示命令正常执行。

Roles 的 Tempalte 的用法及场景在企业中尤为常见，对配置文件的下发及变量替换有着极为灵活便利的支持。但有人会问，如果配置文件因为环境的复杂性需加以一定的逻辑才能生成正确的配置该怎么办呢？6.3 节我们来为大家介绍。

6.2.6 更多复杂的跨平台 Roles

Ansible 支持多平台甚至 Windows (client)。Linux 开源版本多，且各版本间应用管理、进程启动等命令不尽相同，本节为大家介绍 Ansible 如何同时管理多平台。按惯例，我们在完成如下实战场景的同时学习 Ansible 对跨平台的支持。

案例场景：为 Debian、RedHat 两种类型的系统安装 Apache 服务。

步骤 1：编辑 Inventory 文件 /etc/ansible/hosts。

```
[cross-platform]
192.168.37.162 ansible_ssh_user="stanley"
192.168.37.159
```

需要注意的是，192.168.37.162 为 Debian 系统平台，我们默认使用非 root 用户，因为 Ansible 获取主机信息 (gather_facts) 是在执行 Tasks 前进行的，同时 RedHat 系统平台使用的用户是 root，所以我们需要预先定义执行用户，不然会因为使用错误的用户而导致认证失败。

步骤 2：针对不同的系统平台分别编辑 httpd_db、httpd_rh 的 Role。

roles/httpd_db/tasks/{httpd.yml, main.yml} 内容如下：

```
## httpd.yml
---
- name: ubuntu install httpd
  remote_user: stanley
  apt: name=mini-httpd state=present
## main.yml
---
- include: httpd.yml
```

roles/httpd_rh/tasks/{httpd.yml, main.yml} 内容如下：

```
## httpd.yml
---
```

```

- name: centos install httpd
  remote_user: root
  yum: name=httpd state=latest
# # main.yml
---

- include: httpd.yml

```

前面已有多处案例介绍 Roles 目录编排规则，这里不赘述。

步骤 3：编辑 httpd.yml 任务调度文件。

```

---

- name: cross-platform install httpd
  hosts: cross-platform
  roles:
    - { role: httpd_db, when: ansible_os_family == 'Debian' }
    - { role: httpd_rh, when: ansible_os_family == 'RedHat' }

```

最后的安装过程很简单，执行如下命令即可。

```
ansible-playbook httpd.yml
```

关于 Ansible Roles 的使用我们就介绍到这里，本节我们用 Roles 重构了 6.1 节的 Include 代码，并介绍了使用 Roles 重构的优势。在什么样的场景下使用 Roles 更佳需要大家在平时的工作中多加体会。除了 Tasks 在 Roles 的应用外，我们同时也介绍了 Handlers、Files、Templates 的使用，这些在平时的工作中我们都会频繁应用到，需深入掌握。本节我们也提到 Jinja 的一些简单概念，6.3 节我们会带大家更深入地学习 Jinja，掌握了 Jinja 才是深入 Ansible-playbook 的开始。

6.3 Jinja2 实现模板高度自定义

对于较简单的模板变量替换，6.2.5 节内容已经可以满足我们需求，但考虑到模板代码的简洁性和健壮性，本节我们为大家介绍 Jinja2 更多使用技巧。首先，我们来学习 Jinja2 的用法，它和 Linux 系统自带的 BASH Shell 用法一样简单。

6.3.1 Jinja2 For 循环

如 6.2.5 节介绍，变量的提取使用 `{{ variable }}`，`{% statement execution %}` 括起来的内容为 Jinja2 命令执行语句，命令语法如下：

```

{% for item in all_items %}
{{ item }}
{% endfor %}

```

了解其基础用法后，下面我们通过具体案例来进一步掌握 Jinja For 循环用法。

案例场景：为远程主机生成服务器列表，该列表从 192.168.37.201 web01.magedu 开始，到 192.168.37.211 web11.magedu 结束。

在该案例中，手工一条条添加 10 条记录明显不是我们期望的结果，这里需要用 For 循环通过模板批量生成对应的配置文件。我们具体看下对应的 Jinja2 文件该如何编写。

```
{% for id in range(201,211) %}
192.168.37.{{ id }} web{{ "%02d"|format(id-200) }}.magedu
{% endfor %}
```

第 2 行涉及 Python 基础，这里额外解释下。

{{ id }} 提取 for 循环中对应变量 id 的值

"%02d" 调用的是 Python 内置的字符串格式化输出，然后将结果通过管道符 "|" 传递给 format 函数做二次处理，Jinja 支持简单的算法运算符，但不经常用到，这里 format(id-200) 表示 id 的值减去 200

最终执行后结果如下：

```
192.168.37.201 web01.magedu
192.168.37.202 web02.magedu
192.168.37.203 web03.magedu
192.168.37.204 web04.magedu
192.168.37.205 web05.magedu
192.168.37.206 web06.magedu
192.168.37.207 web07.magedu
192.168.37.208 web08.magedu
192.168.37.209 web09.magedu
192.168.37.210 web10.magedu
```

和 Shell 的 For 循环类似，拥有 Bash Shell 基础的人上手 Jinja2 还是很容易的。同理，和 For 循环同等重要的、需要掌握的还有 If 流判断。

6.3.2 Jinja2 If 条件

同写 Bash Shell 脚本一样，具备 If 条件判断的功能非常重要。Jinja 中 If 条件判断的使用格式如下：

```
{% if my_conditional %}
...
{% endif %}
```

我们通过具体的案例来进一步掌握 Jinja If 条件使用。

案例场景：生成 MySQL 配置文件，如果人工指定监听端口则配置为指定端口（预设 1331），否则为默认端口即 3306。

这是我们日常工作中最常遇到且经简化后的业务场景了，从题设大家可以看到这里需要用到 If 判断，具体实现我们来看如下配置。

步骤 1：我们编排目录结构如下。

```
mysqlconf.yml
roles/mysqlconf/
|   templates
|       mycnf.j2
```

该目录结构中，我们只定义了 Templates 而没有定义 Tasks，Ansible 也支持这样的方式，只是 mysqlconf 这个 role 的功能不全而已，但不影响其正常使用。我们本次的 Tasks 调度配置在 mysqlconf.yml 文件中。接下来我们看该文件的配置。

步骤 2：配置 mysqlconf.yml，配置如下。

```
- name      : Mysql conf template
  hosts     : 192.168.37.142
  gather_facts : no
  vars:
    PORT: 1331
    # PORT: false
  tasks:
    - template: src=roles/mysqlconf/templates/mycnf.j2 dest=/etc/mycnf.conf.yml
```

该 YML 文件调用 mysqlconf 这个 role 下的 mycnf.j2 的 tempate，并将其传输至远程主机 192.168.37.142 的 /etc/ 下后，改名为 mycnf.conf.yml。

步骤 3：编辑 roles/mysqlconf/templates/mycnf.j2 模板文件，该配置文件是我们本次 If 条件语句学习的重点。内容如下。

```
{% if PORT %}
bind-address=0.0.0.0:{{ PORT }}
{% else %}
bind-address=0.0.0.0:3306
{% endif %}
```

该代码含义是，如果变量 PORT 被存在，则 bind-address=0.0.0.0:{{PORT 变量的值}}，否则，bind-address=0.0.0.0:3306。

步骤 4：我们来看下效果。执行以下命令。

```
ansible-playbook mysqlconf.yml
```

然后登录远程主机 192.168.37.142，若其内容如下，则表示正常。

```
cat /etc/mycnf.conf.yml
bind-address=0.0.0.0:3306
```

If 的语法至此介绍完毕。

6.3.3 Jinja 多值合并

将多个 Items 合并为一个 List 在日常业务场景中并不少见，如下面我们即将为大家介绍

的这个案例，除了业务场景外，相信大家对用法也有几分眼熟。

```
{% for node in groups["db"] %}
{{ node | join("") }}:5672
{% if not loop.last %}
{% endif %}
{% endfor %}
```

这段代码因为涉及 Jinja 和 Ansible 的内置变量，这里对部分代码做一下解释，

□ 第 1 行代码中 groups 为 Ansible 的内置变量，同类型的内置变量如表 6-1 所示。中括号里的 db 为 Inventory 文件中配置的主机组。

表 6-1 Ansible 部分内置变量

Parameter	Description
hostvars	主机变量名
inventory_hostname	当前 Ansible 可识别的 hosts
group_names	当前主机的所属组
groups	字典数组，数组名，包括：{"all": [...], "web": [...], "ungrouped": [...]}

表中所有内置变量均需了解其功能和用法。

□ 第 2 行使用 Python 内置 join 函数格式化代码输出。

□ 第 3 行 loop.last 为 Jinja2.8 版本的内置变量，同类型及功能如下。

- loop.index: 当前循环的迭代次数（默认从 1 开始）。
- loop.index0: 当前循环的迭代次数（默认从 0 开始）。
- loop.revindex: 到循环结束需要迭代的次数（默认从 1 开始）。
- loop.revindex0: 到循环结束需要迭代的次数（默认从 0 开始）。
- loop.first: 如果是第一次迭代，为 True。
- loop.last: 如果是最后一次迭代，为 True。
- loop.length: 序列中的项目数。
- loop.depth: 显示渲染的递归循环的层级数（默认从 1 开始）。
- loop.depth0: 显示渲染的递归循环的层级数（默认从 0 开始）。
- loop.cycle: 在一串序列间周期取值的辅助函数。更详细资料的请参考 Jinja 的官网文档^①。

我们看下该命令执行方式及结果，编排目录如下：

```
join.yml
roles/join/
├── templates
└── list.j2
```

① Jinja 官方地址：<http://jinja.pocoo.org/>。

roles/join/templates/list.j2 内容如下:

```
{% for node in groups["db"] %}
{{ node | join("") }}:5672
{% if not loop.last %}
{% endif %}
{% endfor %}
```

编辑 join.yml 内容如下:

```
- name      : Join loop list Combining multiple values
  hosts     : db
  gather_facts : no
  vars:
    PORT: 1331
```

tasks:

```
- template: src=roles/join/templates/list.j2 dest=/data/list.txt
```

roles:

```
- { role: join }
```

执行命令如下:

```
ansible-playbook join.yml
```

登录远程主机 192.168.37.142 查看 /data/list.txt, 内容如下为正常。

```
192.168.37.142:5672
192.168.37.159:5672
```

如果第 2 行代码变更为如下:

```
{{ node | join("-") }}:5672
```

那输出的结果会是什么? 留给大家自己测试, 希望能加深大家的理解。

本节接触的新内容有 join 实现多值合并、Jinja 内置变量 loop 循环体、Ansible 内置变量 groups。更多内容请参考 Jinja 官网 <http://jinja.pocoo.org/> 和 Ansible 官网 <http://docs.ansible.com/>。

6.3.4 Jinja default() 设定

精通程序编码的人皆知, default() 默认值的设定有助于程序的健壮性和简洁性。所幸 Jinja 也支持该功能, 大家还记得 6.3.2 节生成 MySQL 配置文件中的端口定义吗? 如果指定则 PORT=1331, 否则 PORT=3306。我们将该案例改造为使用 default()。

编辑 roles/mysqlconf/templates/mycnf.j2, 内容如下:

```
bind-address=0.0.0.0:{{ PORT | default(3306) }}
```

这种方式是否更为简洁呢？原来的 5 行代码通过 `default()`，只需要一行就能实现我们所希望的功能。另外，为了帮助大家更好地掌握 `default()` 的使用，我们摘录网友的一段代码供大家参考学习，同时大家也可以思考如何修改优化该代码。

```
{% for item in NETWORK_INTERFACES %}
auto {{ item.vlan_name }}
iface {{ item.vlan_name }} inet static
    address {{ item.ip_net }}.{{ HOST_IP_OCTET }}
    netmask 255.255.255.0
    network {{ item.ip_net }}.0
    broadcast {{ item.ip_net }}.255
    pre-up ip link add link {{ item.dev }} name {{ item.vlan_name }} type vlan
    id {{ item.vlan_tag }}
    pre-up ip link set dev {{ item.vlan_name }} mtu 9000
    pre-up ethtool -K {{ item.vlan_name }} gro off
    post-down ip link delete {{ item.vlan_name }}
{% if item.gateway is defined %}
    post-up ip route add {{ item.ip_net }}.0/24 dev {{ item.vlan_name }} table {{ item.
rt_table }}
    post-up ip route add default via {{ item.gateway }} table {{ item.rt_table }}
    post-up ip rule add from {{ item.ip_net }}.{{ HOST_IP_OCTET }} table {{ item.
rt_table }}
    pre-down ip rule delete from {{ item.ip_net }}.{{ HOST_IP_OCTET }} table {{ item.
rt_table }}
    pre-down ip route delete default via {{ item.gateway }} table {{ item.rt_table }}
    pre-down ip route delete {{ item.ip_net }}.0/24 dev {{ item.vlan_name }} table {{
item.rt_table }}
{% endif %}
{% if item.default_gateway is defined %}
    post-up ip route add default via {{ item.default_gateway }}
    pre-down ip route delete default via {{ item.default_gateway }}
    dns-search baremetal.{{ DOMAIN }} {{ DOMAIN }}
    dns-nameservers 10.0.0.{{ HOST_IP_OCTET }} 8.8.8.8
{% endif %}
{% endfor %}
```

如上代码看似长，但其实不难，请大家思考是否有优化改造的余地。至此 Jinja 的语法格式及技巧已经介绍完毕。接下来我们会通过更多的实战来巩固大家的学习成果。

6.3.5 Ansible 结合 Jinja2 生成 Nginx 配置

Nginx 是目前最流行的 WebServer 应用，本节为大家介绍 Nginx 配置文件的生成。

案例场景：为 2 台 Nginx Proxy、1 台 Nginx Web 通过一套模板生成对应的配置。

步骤 1：编排目录如下。

```
nginxconf.yml
roles/nginxconf/
└── tasks
```

```
|  └── file.yml
|  └── main.yml
└── templates
    ├── nginx.conf.j2
    └── vars
        └── main.yml
```

目录的编排已多次提及，这里不赘述。

步骤 2: 编辑 nginxconf role 的 tasks 调度文件 roles/nginxconf/tasks/{file.yml, main.yml}。

编辑 file.yml, 定义 nginxconf role 的一个功能集 (一个文件一个功能集)。

```
- name: nginx.conf.j2 template transfer example
  template: src=nginx.conf.j2 dest=/etc/nginx/nginx.conf.template
```

编辑 main.yml，定义任务功能集合、nginxconf role 功能集入口。

```
- include: file.vml
```

步骤 3：这是最重要的一步，定义 nginxconf role 的模板文件 roles/nginxconf/templates/nginx.conf.j2，该模板的灵活性将直接影响 Ansible-playbook 的代码行数和整体 Playbook 的灵活性健壮性。该模板文件将被替换变量后生成最终的 Nginx 配置文件。

```
{% if nginx_use_proxy %}
{% for proxy in nginx_proxies %}
upstream {{ proxy.name }} {
    # server 127.0.0.1:{{ proxy.port }};
    server {{ ansible_eth0.ipv4.address }}:{{ proxy.port }};
}
{% endfor %}
{% endif %}
server {
    listen 80;
    server_name {{ nginx_server_name }};
    access_log off;
    error_log /dev/null crit;
    rewrite ^ https://$server_name$request_uri? permanent;
}
server {
    listen 443 ssl;
    server_name {{ nginx_server_name }};
    ssl_certificate /etc/nginx/ssl/{{ nginx_ssl_cert_name }};
    ssl_certificate_key /etc/nginx/ssl/{{ nginx_ssl_cert_key }};

    root {{ nginx_web_root }};
    index index.html index.html;
```



```

{% if nginx_use_auth %}
    auth_basic "Restricted";
    auth_basic_user_file /etc/nginx/{{project_name}}.htpasswd;
{% endif %}

{% if nginx_use_proxy %}
{% for proxy in nginx_proxies %}

    location {{ proxy.location }} {
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-Proto http;
        proxy_set_header X-Url-Scheme $scheme;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_set_header X-NginX-Proxy true;
        proxy_redirect off;
        proxy_pass http://{{ proxy.name }};
        break;
    }
{% endfor %}
{% endif %}

{% if nginx_server_static %}
    location / {
        try_files $uri $uri/ =404;
    }
{% endif %}
}

```

这里需要为大家显示完整的代码，以方便整体概览，代码稍多但不复杂，考虑到篇幅不再解释，难以理解的请从 6.3.1 节开始按序阅读。

步骤 4：编辑 nginxconf role 的变量文件 roles/nginxconf/vars/main.yml。

```

nginx_server_name: www.magedu.com
nginx_web_root: /opt/magedu/
nginx_proxies:
- name: suspicious
  location: /
  port: 2368
- name: suspicious-api
  location: /api
  port: 3000

```

该变量文件需要关注的是 `nginx_proxies` 定义的变量组，其下的变量列表通过 `for` 循环读取后可以通过 “.” 来引用，即如下 `proxy.name` 这样的引用方式。

```
{% for proxy in nginx_proxies %}

upstream {{ proxy.name }} {
    # server 127.0.0.1:{{ proxy.port }};
}
```

步骤 5: 编辑总调度文件 nginxconf.yml。

```
- name: Nginx Proxy Server's Conf Dynamic Create
  hosts: "192.168.37.130:192.168.37.158"
  vars:
    nginx_use_proxy: true
    nginx_ssl_cert_name: ifa.crt
    nginx_ssl_cert_key: ifa.key
    nginx_use_auth: true
    project_name: suspicious
    nginx_server_static: true
    gather_facts: true

  roles:
    - { role: nginxconf }

- name: Nginx WebServer's Conf Dynamic Create
  hosts: 192.168.37.159
  vars:
    nginx_use_proxy: false
    nginx_ssl_cert_name: ifa.crt
    nginx_ssl_cert_key: ifa.key
    nginx_use_auth: false
    project_name: suspicious
    nginx_server_static: false
    gather_facts: no

  roles:
    - { role: nginxconf }
```

在 nginxconf.yml 文件中, 同样我们也定义 nginx_use_proxy、nginx_ssl_cert_name、nginx_ssl_cert_key、nginx_use_auth、project_name、nginx_server_static 等变量, 同时不同类型的主机定义的不同变量生成的配置文件也不尽相同, Ansible 的灵活性可见一斑。

步骤 6: 验证结果。

执行命令如下:

```
ansible-playbook nginxconf.yml
```

然后登录到 NginxWeb 192.168.37.159 查看 /etc/nginx/nginx.conf.template 配置文件, 类似如下输出表示达到我们的预期。

```
server {
    listen 80;
    server_name www.magedu.com;
```

```

access_log off;
error_log /dev/null crit;
rewrite ^ https://$server_name$request_uri? permanent;
}

server {
    listen 443 ssl;
    server_name www.magedu.com;
    ssl_certificate /etc/nginx/ssl/ifa.crt;
    ssl_certificate_key /etc/nginx/ssl/ifa.key;
    root /opt/magedu/;
    index index.html index.html;
}

```

同理，登录到 NginxProxy 主机，查看其中一台 Proxy 的 nginx.conf.template 配置，类似如下输出表示达到我们的预期。

```

upstream suspicious {
    # server 127.0.0.1:2368;
    server 192.168.37.130:2368;
}

upstream suspicious-api {
    # server 127.0.0.1:3000;
    server 192.168.37.130:3000;
}

server {
    listen 80;
    server_name www.magedu.com;
    access_log off;
    error_log /dev/null crit;

    rewrite ^ https://$server_name$request_uri? permanent;
}

server {
    listen 443 ssl;
    server_name www.magedu.com;

    ssl_certificate /etc/nginx/ssl/ifa.crt;
    ssl_certificate_key /etc/nginx/ssl/ifa.key;

    root /opt/magedu/;
    index index.html index.html;

    auth_basic "Restricted";
    auth_basic_user_file /etc/nginx/suspicious.htpasswd;
}

```

```

location / {
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-Proto http;
    proxy_set_header X-Url-Scheme $scheme;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_set_header X-NginX-Proxy true;
    proxy_redirect off;

    proxy_pass http://suspicious;
    break;
}

location /api {
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-Proto http;
    proxy_set_header X-Url-Scheme $scheme;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_set_header X-NginX-Proxy true;
    proxy_redirect off;

    proxy_pass http://suspicious-api;
    break;
}

location / {
    try_files $uri $uri/ =404;
}
}

```

同样的模板，通过简单的 if 和变量设置，就可以完成不同类型主机的 Nginxconf 配置，所以在了解 Ansible 强大的模板功能的同时，也显示出模板质量的重要性。

6.3.6 Ansible 结合 Jinja2 生成 Apache 多主机配置

6.3.5 节介绍了 Nginx 配置的生成，本节介绍 Apache 多主机配置的生成。因本节不涉及新知识点，以纯实践为主，考虑篇幅，我们尽可能地简洁明了地介绍。

案例场景：通过 Ansible 的 Jinja 模板，生成如下的 Apache 多主机配置。

```

NameVirtualHost *:80

<VirtualHost *:80>
    ServerName apache.magedu.com
    DocumentRoot /data/magedu/
    <Directory "/data/magedu/">

```



```

    AllowOverride All
    Options -Indexes FollowSymLinks
    Order allow,deny
    Allow from all
  </Directory>
</VirtualHost>

<VirtualHost *:80>
  ServerName apache.magedu.otherdomain.com
  DocumentRoot /data/otherdomain/
  ServerAdmin stanley@magedu.com
  <Directory "/data/otherdomain/">
    AllowOverride All
    Options -Indexes FollowSymLinks
    Order allow,deny
    Allow from all
  </Directory>
</VirtualHost>

```

步骤 1: 编排目录结构如下。

```

roles/apacheconf/
├── tasks
│   ├── file.yml
│   └── main.yml
├── templates
│   └── apache.conf.j2
└── vars
    └── main.yml

```

步骤 2: 编辑 apacheconf role 的 tasks 调度文件 roles/apacheconf/tasks/{file.yml, main.yml}。

编辑 file.yml, 内容如下:

```

---

- name: Apache.conf.j2 tempalte transfer example
  template: src=apache.conf.j2 dest=/etc/httpd/apache.conf.template

```

编辑 main.yml, 内容如下:

```

---

- include: file.yml

```

步骤 3: 定义 apacheconf role 的模板文件 roles/apacheconf/templates/apache.conf.j2。内容如下:

```

NameVirtualHost *:80

{% for vhost in apache_vhosts %}
<VirtualHost *:80>

```

```

    ServerName {{ vhost.servername }}
    DocumentRoot {{ vhost.documentroot }}
    {% if vhost.serveradmin is defined %}
        ServerAdmin {{ vhost.serveradmin }}
    {% endif %}
    <Directory "{{ vhost.documentroot }}">
        AllowOverride All
        Options -Indexes FollowSymLinks
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
{% endfor %}

```

步骤 4: 编辑 apacheconf role 的变量文件 roles/apacheconf/vars/main.yml。

```

---
apache_vhosts:
  - {servername: "apache.magedu.com", documentroot: "/data/magedu/"}
  - {servername: "apache.magedu.otherdomain.com", documentroot: "/data/otherdomain/",
    serveradmin: "stanley@magedu.com"}

```

步骤 5: 编辑总调度文件 apacheconf.yml。

```

- name: Apache WebServer's Conf Dynamic Create
  hosts: 192.168.37.159
  gather_facts: no

  roles:
    - { role: apacheconf }

```

步骤 6: 验证结果。

执行命令如下:

```
ansible-playbook apacheconf.yml
```

可以登录到 192.168.37.159, 查看 /etc/httpd/apache.conf.template 是否和我们预期的一样。

6.3.7 Jinja2 动态变量配置及架构优化

至此, 所有的 Jinja 技能点均已介绍完毕。本章的最后我们来研究架构优化。众所周知, 好的架构不一定是复杂的, 但一定是能很好承载当前业务需求的。Ansible 一向以简单著称, 当然也不建议写出很长很复杂的 Playbook。同时经验也告诉我们, 如果真的遇到 Playbook 很长的情况, 还是建议停下来认真理头绪和思路。下面的案例供大家在架构优化的过程中参考学习。

案例场景: 我们希望将变量通过命令行传递给 Playbook, 当 param1 定义的时候, myvariable=

param1 即 value1。

当 param1、param2 定义的时候，myvariable=value1, value2。

当 param1、param2、param3 定义的时候，myvariable=value1, value2, value3。

按常规思路，通过如下命令即能完成功能所需：

```
ansible-playbook playbook.yml -e "param1=value1 param2=value2 param3=value3"
```

那么，我们按现有题目原设思路的解法如下。

步骤 1：编排 myrole 目录。

```
myrole.yml
roles/myrole/
├── templates
│   └── myvar.j2
└── vars
    └── main.yml
```

步骤 2：编辑模板文件 roles/myrole/templates/myvar.j2 和创建空变量文件。

```
myvariable: {{ myvariable }}
```

创建空变量文件。

```
mkdir roles/myrole/vars/ && touch roles/myrole/vars/main.yml
```

步骤 3：编辑总调度文件 myrole.yml。

```
- name      : Test var
  hosts     : 192.168.37.142
  gather_facts : no
  vars:
    myvariable: "{{ [param1|default(''), param2|default(''), param3|default('')] | join(',') }}"
    # myvariable: "{{ [param1, param2, param3] | reject('undefined') | join(',') }}"
  tasks:
    - template: src=roles/myrole/templates/myvar.j2 dest=/data/main.yml
    - debug:
        var=myvariable

roles:
  - { role: myrole }
```

执行结果 Ansible 动态变量如图 6-4 所示。

这段代码摘自网络，其最终的执行结果无可厚非，确实达到最终目的，但出现该场景却是值得反思的。Ansible 虽然提供了多值合并的 join 用法，但本质上是为了功能的完整性，这和 Ansible 简洁明了的风格是相违背的。我们换一种方法来处理这个问题，虽然不是最好的，但在思路风格上相对更贴合 Ansible 的核心精神。

```
[root@linux1st fab2ansible]# ansible-playbook myrole.yml -e "param1=value1 param2=value2"
PLAY [Test var] *****
TASK: [template src=roles/myrole/templates/myvar.j2 dest=/data/main.yml] *****
ok: [192.168.37.142]
TASK: [debug var=myvariable] *****
ok: [192.168.37.142] => {
  "var": {
    "myvariable": "value1,value2,"
  }
}
PLAY RECAP *****
192.168.37.142 : ok=2 changed=0 unreachable=0 failed=0

[root@linux1st fab2ansible]# ansible-playbook myrole.yml -e "param1=value1 param2=value2 param3=value3"
PLAY [Test var] *****
TASK: [template src=roles/myrole/templates/myvar.j2 dest=/data/main.yml] *****
changed: [192.168.37.142]
TASK: [debug var=myvariable] *****
ok: [192.168.37.142] => {
  "var": {
    "myvariable": "value1,value2,value3"
  }
}
PLAY RECAP *****
192.168.37.142 : ok=2 changed=1 unreachable=0 failed=0
```

图 6-4 Ansible 动态变量

```
- name: Test var
  hosts: 192.168.37.142
  gather_facts: no
  vars:
    myvariable: false
  tasks:
    - name: param1
      set_fact:
        myvariable: "{{param1}}"
      when: param1 is defined
    - name: param2
      set_fact:
        myvariable: "{{ param2 if not myvariable else myvariable + ',' + param2 }}"
      when: param2 is defined
    - name: param3
      set_fact:
        myvariable: "{{ param3 if not myvariable else myvariable + ',' + param3 }}"
      when: param3 is defined
    - name: default
      set_fact:
        myvariable: "default"
```



```

when: not myvariable

- debug:
    var=myvariable

- template: src=roles/myrole/templates/myvar.j2 dest=/data/main.yml

roles:
  - { role: myrole }

```

该方法可能不是最好的，但在编码风格上更贴合 YAML 风格，代码整体简单易懂，更易于维护和扩展。这两种处理方式只是给用户评价代码好坏提供参考标准和优化方向，复习已有的代码和思路，思考为什么出现这样的场景是推荐的办法。

6.4 Ansible Galaxy

从第6章开始，大家会发现我们的代码愈加复杂，项目越大对 Roles 的依赖越明显，同时对使用者技术和架构能力的要求也越来越高。我们尽力优化代码，尽力使其看起来简单灵活，但又不失去功能的强大，这个过程需要每个人投入心血和思考，知识需要积累，精髓值得传承。Galaxy 为我们提供了这样的平台，本节我们来为大家介绍 Galaxy 的使用。

此处的 Galaxy 并非手机，而是 Ansible 官方 Roles 分享平台，在 Galaxy 平台所有人可以免费上传或下载 Roles，在这里好的技巧、思想、架构得以积累和传播。这也是推荐大家尽可能使用 Roles 的原因，在 Roles 中有无限的资源可供调用，同时你也很有可能因为自己上传的精品 Roles，一夜之间受到大家的关注。

6.4.1 Ansible-galaxy 命令用法

Ansible-galaxy 是 Ansible 系列工具之一，主要用于管理 galaxy.ansible.com 的 Roles，默认下载的 Roles 存放于 /etc/ansible/roles 目录下，可在 /etc/ansible/ansible.cfg 自定义存放目录。其命令用法如下。

1) 获取命令用法。

```

ansible-galaxy --help
Usage: ansible-galaxy [init|info|install|list|remove] [--help] [options] ...

```

Options:

```
-h, --help show this help message and exit
```

See 'ansible-galaxy <command> --help' for more information on a specific command.

2) Ansible-galaxy 有 init、info、install、list、remove 方法，用法如下。

init: 创建空 Roles, 用于向 galaxy.ansible.com 上传代码。Usage: ansible-galaxy init [options] role_name

info: 显示 Roles 的版本号、作者、下载次数、Commit 信息、版本依赖等详细信息。Usage: ansible-galaxy info [options] role_name[,version]。如显示 manala.mysql 这个 Role 的详细信息, 命令用法为: ansible-galaxy info manala.mysql

install: 安装 Roles, Usage: ansible-galaxy install [options] [-r FILE | role_name(s)[,version] | scm+role_repo_url[,version] | tar_file(s)]。如 ansible-galaxy install manala.mysql

list: 列出本地已安装的所有 Roles。Usage: ansible-galaxy list [role_name]

remove: 移除本地指定的 Roles。Usage: ansible-galaxy remove role1 role2 ...

了解 Galaxy 的使用方法后, 我们继续介绍 Galaxy 的具体使用。

6.4.2 使用 Galaxy

(1) 下载 Roles

[https://galaxy.ansible.com/explore/#/](https://galaxy.ansible.com/explore#/) 多维度划分 Roles 的下载使用情况, 如最流行、下载量最多、最新上架、最多贡献者等。https://galaxy.ansible.com/list#/roles?page=1&page_size=10 可根据所需关键字搜索。Galaxy 上的 Roles 命名规范遵循 username.rolename。所以下载我们使用:

```
ansible-galaxy install username.rolename
```

如需同时下载多个 Roles, 可将所需 Roles 写入配置文件后, 使用 -r 批量下载 Roles。如:

```
# roles.txt
user1.role1,v1.0.0
user2.role2,v0.5
user2.role3
```

使用 Ansible-galaxy 批量安装 Roles。

```
ansible-galaxy install -r roles.txt
```

另外, 从 Ansible 1.8 后开始, Galaxy 支持从其他源下载、指定下载路径、Roles 安装命名等高级下载功能, 该功能可以通过编写 YAML 文件实现。官方案例摘录如下:

```
# install_roles.yml

# 从 galaxy 下载
- src: yatesr.timezone

# 从 GitHub 下载
- src: https://GitHub.com/bennojoy/nginx

# 从 GitHub 下载安装到本机的相对路径下
- src: https://GitHub.com/bennojoy/nginx
  path: vagrant/roles/
```

```
# 从 GitHub 下载指定版本的源码
- src: https://GitHub.com/bennojoy/nginx
  version: master
  name: nginx_role

# 从 Web 下载服务器, 打包为 tar.gz 压缩包形式的源码
- src: https://some.webserver.example.com/files/master.tar.gz
  name: http-role

# 如果 bitbucket 网站可用, 从 bitbucket 站点下载
- src: git+http://bitbucket.org/willthames/git-ansible-galaxy
  version: v1.4

# 从 bitbucket 网站下载
- src: http://bitbucket.org/willthames/hg-ansible-galaxy
  scm: hg
```

安装 Roles:

```
ansible-galaxy install -r install_roles.yml
```

(2) 删除从 Galaxy 下载的 Roles

删除的方式很简单, 可以手动至 `/etc/ansible/roles/` 删除指定的目录即可。Ansible-galaxy 也提供了专门的删除命令: `ansible-galaxy remove role1 role2 ...`。

如先来查看本地已下载的所有 Roles。

```
ansible-galaxy list
- manala.mysql, master
- hectcastro.nginx, 0.1.1
```

删除 manala.mysql、hectcastro.nginx。

```
ansible-galaxy remove manala.mysql hectcastro.nginx
```

(3) 上传分享自己的优秀 Roles

想上传自己的 Roles 需要满足如下几个条件。

☐ 正常访问 Internet, 能正常打开 <http://www.GitHub.com>、<http://galaxy.ansible.com>。

☐ 开通了 GitHub 账户。

☐ 编写好遵循 Galaxy 标准规范的 Roles。

详情请参考官网 <https://galaxy.ansible.com/intro#share>。

关于 Galaxy 部分我们介绍得相对粗略, 一方面因为 Galaxy 的使用方式国人还不是那么习惯, 考虑到安全等原因, 下载后的 Roles 每行代码都需要研读一遍, 所以多数国人更多参考的是其用法技巧, 思路和架构依然以自学为主。关于代码语法大家也可以参考 <https://GitHub.com/ansible/ansible-examples>。另一方面国人对 Galaxy 的使用场景多数还停留在

下载的层面，在大家努力学习国外优秀代码的同时，也希望能有更多的国产代码出现在 Galaxy 上。

6.5 本章小结

关于 Playbook 在 Ansible 的应用，通过本节内容的学习相信大家也有所体会。本节中我们接触了提高代码复用率的工具 Include、模块化代码块工具 Roles、高度自定义配置模板工具 Jinja、Roles 代码共享学习平台 Galaxy，既了解到如何使用 Include、Roles，也了解到如何优化自己的代码段，同时也知晓如何获取优化的 Roles 代码段及如何分享自己的优秀代码段。本章是 Ansible 用法最为核心的技巧内容了，所以请务必深入掌握。关于 Ansible 的使用技巧性内容到本章为止绝大多数均已悉数介绍，本书后半部分的内容重点为 Ansible 与业内主流技术的结合使用，以及 Web 自动化开发，接下来我们会为大家一一呈现。

Inventory 文件扩展

经过前面诸多章节的学习，相信大家对 Inventory 文件（默认 /etc/ansible/hosts）的基本定义和用法都有了大致的了解。下面我们再简单回顾一下，请看下面这个 Inventory 文件。

```
# Inventory 文件 /etc/ansible/hosts
# 主机组使用中括号“[]”来定义，接下来，每一行定义一台组内的主机
[myweb]
www.myweb.com
10.23.1.123
```

当需要对整个主机组 myweb 里面的主机进行操作时，像下面这样，直接在 Playbook 中使用主机组名就可以了。

```
---
- hosts: myweb
  tasks:
    [...]
```

在 Ad-Hoc 命令行模式下，对主机组或某台主机进行操作则更为简单，直接将在 Inventory 中定义过的主机组名或主机名跟在 ansible 命令后面即可，比如：

```
ansible myweb -a "free -m"
```

回顾结束，接下来我们来看一些实际生产环境中的案例。

7.1 Inventory 文件实战

实际生产环境中，根据业务量的规模差异，Inventory 文件中的主机数量会从几十台到上

百台不等。通常这些主机会按照其所服务的应用类型进行分组，比如 database、webserver 和 caching 组等。

下面我们来看一个现实生产中的案例。该案例中，我们使用 Check.in 服务来对服务器的 uptime 进行监控，其 Inventory 文件内容如代码清单 7-1 所示。

代码清单 7-1 Inventory 文件内容

```

1 # Check.in 服务器端主机组
2 [servercheck-web]
3 www1.servercheck.in
4 www2.servercheck.in
5
6 [servercheck-web:vars]
7 ansible_ssh_user=servercheck_svc
8
9 [servercheck-db]
10 db1.servercheck.in
11
12 [servercheck-log]
13 log.servercheck.in
14
15 [servercheck-backup]
16 backup.servercheck.in
17
18 [servercheck-nodejs]
19 atl1.servercheck.in
20 atl2.servercheck.in
21 nyc1.servercheck.in
22 nyc2.servercheck.in
23 nyc3.servercheck.in
24 ned1.servercheck.in
25 ned2.servercheck.in
26
27 [servercheck-nodejs:vars]
28 ansible_ssh_user=servercheck_svc
29 foo=bar
30
31 # 按操作系统类型对主机组进行分组
32 [centos:children]
33 servercheck-web
34 servercheck-db
35 servercheck-nodejs
36 servercheck-backup
37
38 [ubuntu:children]
39 servercheck-log

```

一眼看过去，这个 Inventory 文件让人有点不知该如何下手，但是当我们把它拆开来看的话，会发现它其实表述的是一幅简单的系统架构图。

图 7-1 为 Check.in 服务器架构图，可将其与 Inventory 文件对照。

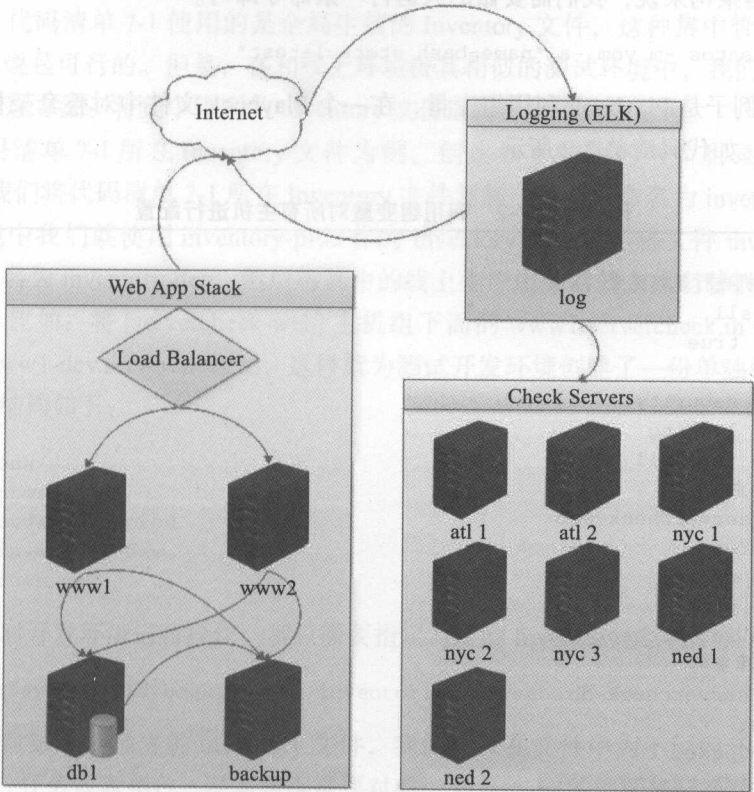


图 7-1 Check.in 服务器架构图

在上述 Inventory 文件（代码清单 7-1）中：

第 1 ~ 19 行是对主机组进行定义（有些主机组中只有一台主机），这里定义的主机组名称就可以在 Ansible 命令和 Playbook 中被直接使用。第 6 ~ 7 行和第 27 ~ 29 行分别为主机组 servercheck-web 和 servercheck-nodejs 定义了组变量，这些组变量对相关主机组内的所有主机生效。

第 31 ~ 39 行对主机组又重新按不同系统类型进行分组，主机组可以包含主机组，主机的变量可以通过继承关系，继承到最高等级的组的变量。定义主机组之间的继承关系使用“children”来表示。由此也可以看出，Inventory 文件中主机组的划分是十分灵活的。

在使用 Ansible 的过程中，这种灵活的分组方式为 Ansible 提供了非常大的灵活性。比如，现在这套架构中，我们要为所有的 CentOS 系统打一个补丁，只需要对 CentOS 主机组进行操作即可，不需要再逐台登录主机进行操作，也避免了在其他非 CentOS 系统的主机上进行判断和操作，从而更具效率。比如 2014 年 9 月曝出的 BASH 安全漏洞 Shellshock，大漏洞

被曝出几小时后，补丁文件就发布了出来，只需将 BASH 升级到最新版即可。对于我们这套 Check.in 服务器架构来说，我们需要做的只运行一条命令即可。

```
ansible centos -m yum -a "name=bash state=latest"
```

下面一个例子是 Playbook 利用组变量，在一个 Playbook 文件中对整套架构中的所有主机进行了配置，如代码清单 7-2 所示。

代码清单 7-2 利用组变量对所有主机进行配置

```
---
# 对所有主机进行基础配置
- hosts: all
  sudo: true
  roles:
    - security
    - logging
    - firewall
# 配置 web 主机
- hosts: servercheck-web
  roles:
    - nginx
    - php
    - servercheck-web
# 配置数据库主机
- hosts: servercheck-db
  roles:
    - pgsql
    - db-tuning
# 配置日志主机
- hosts: servercheck-log
  roles:
    - java
    - elasticsearch
    - logstash
    - kibana
# 配置备份主机
- hosts: servercheck-backup
  roles:
    - backup
# 配置 Node.js 主机
- hosts: servercheck-nodejs
  roles:
    - servercheck-node
```

构建服务器基础设施管理的 Playbook 方法有很多，这只是其中一种，在随后的章节中我们会继续探讨更多的方法。对于结构比较简单的服务器架构来说，代码清单 7-2 的方法足矣，同时，它还具有很强的可扩展性。

7.2 独立的 Inventory 文件

上节中，代码清单 7-1 使用的是全局生效的 Inventory 文件，这种集中管理的方法对线上生产环境来说是可行的。但是，在和线上环境极其相似的测试环境中，我们需要另外一种 Inventory 的管理方法：将集中管理的 Inventory 文件进行分隔，独立管理。

还以代码清单 7-1 所在 Inventory 文件为例，创建如下结构的两个目录：servercheck/inventories，我们将代码清单 7-1 所在 Inventory 文件复制一份并重命名为 inventory-prod，这表明生产环境中我们就使用 inventory-prod 作为 Inventory 文件。再将文件 inventory-prod 复制一份并重命名为 inventory-dev，然后将其中的线上生产服务器的主机名替换为测试开发环境的主机名，比如：将 [servercheck-web] 主机组下面的 www1.servercheck.in 改为与其对应的测试主机 www1-dev.servercheck.in，这样就为测试开发环境创建了一份单独的 Inventory 文件。最后目录结构如下：

```
servercheck/
  inventories/
    inventory-prod
    inventory-dev
  playbook.yml
```

现在要想对开发环境进行操作，则只需要指定对应的 Inventory 文件就可以了。比如：

```
ansible-playbook playbook.yml -i inventories/inventory-dev
```

此外，针对第一个独立的 Inventory 文件，我们可以在文件中为不同的环境指定不同的 Inventory 变量、任务或者角色，甚至根据需要对整个 Inventory 的架构进行变动都是可以的。

7.3 Inventory 变量

在本书第 5 章中，我们曾介绍过如何通过 Inventory 文件为某个主机组或某个单独主机设置变量的方法。本节我们将介绍另外一些通过 Inventory 设置变量的方法。

我们先来回顾一下在 Inventory 文件中为主机和主机组定义变量的简单例子。代码如下所示：

```
[www]
# 为主机单独定义变量
www1.example.com ansible_ssh_user=johndoe
www2.example.com
```

```
[db]
db1.example.com
db2.example.com
```

为一组主机定义变量，这些变量对组内所有主机生效

```
[db:vars]
ansible_ssh_port=5222
database_performance_mode=true
```

通常，我们建议不要在静态的 Inventory 文件中定义过多的变量，因为这样定义的变量不识别度低，而且维护起来也比较麻烦，尤其是在一行内为一台主机定义多个变量的时候。

幸运的是，Ansible 为用户提供一种弹性很高且易于维护的变量管理方法。

7.3.1 host_vars 目录

在很多项目中，同一主机组中的各个主机可能由于自身硬件性能的差异或所运行服务的不同，对同一项性能指标有着不同的要求。比如在 Apache Slor 集群中，我们有一个名为 slor 的主机组，里面各主机对内存有着不同的需求。我们可以使用 host_vars 目录对每一台主机进行变量设置。host_vars 目录可以将 Hosts 文件一同放置在 /var/ansible 目录下，也可以与 Playbook 文件放在同一个目录下，host_vars 目录内放置和主机同名的 YAML 文件，用来为主机设置变量。

下面我们来看一个简单的 host_vars 目录的应用实例。我们当前有如下的目录结构：

```
hostedapachesolr/
  host_vars/
    nyc1.hostedapachesolr.com
  inventory/
    hosts
  main.yml
```

本例中 inventory 目录下的文件 hosts 定义主机组，其内容如下：

```
[solr]
nyc1.hostedapachesolr.com
nyc2.hostedapachesolr.com
jap1.hostedapachesolr.com
...
[log]
log.hostedapachesolr.com
```

在 Ansible 运行时，Ansible 会搜索 hostedapachesolr/host_vars/nyc1.hostedapachesolr.com 或者 hostedapachesolr/inventory/host_vars/nyc1.hostedapachesolr.com（本例中未使用该文件），在这两个文件中定义的变量只对文件名所对应的主机名生效，并且将覆盖在其他任何 Playbook 和 Role 中定义的同名变量的值。

文件 nyc1.hostedapachesolr.com 的内容如下：

```
---
tomcat_xmx: "1024m"
```

默认情况下，tomcat_xmx 的值为 640m，我们在 nyc1.hostedapachesolr.com 进行的设置，

将会覆盖其默认值，使其最终结果为 1024m。

使用 `host_vars` 目录的方法来管理和定义主机变量非常便于维护，并且由于文件名是由主机名命名的 YAML 文件，所以维护起来也不容易搞混。

7.3.2 group_vars 目录

`group_vars` 目录管理组变量的方法与 `host_vars` 目录非常相似，存放路径也是在 `/etc/ansible` 目录下或者与所要执行的 Playbook 相同的目录下，用于定义组变量的文件也要使用 YAML 语法，且文件应以主机组名来命名。

我们继续使用上面的例子，只是在原有的目录结构中加了一个 `hostedapachesolr/group_vars` 目录。结构如下所示：

```
hostedapachesolr/
  group_vars/
    solr
  host_vars/
    nycl.hostedapachesolr.com
  inventory/
    hosts
  main.yml
```

在文件 `group_vars/solr` 中，使用 YAML 语法为主机组 `slor` 定义组变量，内容如下：

```
---
do_something_amazing=true
foo=bar
```

7.4 动态 Inventory

在大多数情况下，静态 Inventory 文件可以很好地描述主机间的关系。尤其是服务器规模不大的情况下，即便是手动来编辑、更新 Inventory 文件也非常方便快捷。

然而，我们所生活的时代是云计算和大规模集群的时代。在实际生产应用中，经常会遇到业务的快速发展或者流量的急剧增加等情况，需要在短时间内向架构中添加几十台甚至上百台服务器来提高整个架构的处理能力。这个时候，手动管理 Inventory 文件不仅没有效率，而且非常乏味。

此时，动态 Inventory 应运而生。Ansible 通过调用第三方脚本来动态地配置 Inventory 文件。目前，一些知名的云主机供应商，如亚马逊 AWS、Cobbler、gitalOcean、Lnode、OpenStack 等，提供了现成的脚本供 Ansible 直接调用，其具体的用法在对应的平台上都有详尽的官方文档说明，这里我们不赘述。接下来我们将借助实际案例，详细介绍如何自行开发动态 Inventory 文件的脚本。

Ansible 启用动态 Inventory 的机制是通过调用外部脚本（任何脚本都可以，二进制文件

也可以，只要运行结果返回的是 JSON 串就行）生成指定格式的 JSON 串。Ansible 可以对 JSON 格式的字符串进行解析，并最终将其转化为 Ansible 可用的 Inventory 文件格式。所以，所谓的动态 Inventory 文件脚本开发，其实就是编写脚本根据具体环境将主机信息及关系（这些数据可以通过抓取数据库，调用外部 API 或者直接读取文件获得）以 JSON 格式来表示出来，并将其做为脚本输出结果传给 Ansible。

需要注意的是，用于生成 JSON 代码的脚本必须支持两个选项：`--list` 和 `--host`。

❑ `--list`：返回所有的主机组信息，每个组都应该包含字典形式的主机列表、子组列表，如果需要的话还应该组变量，最简单的信息是只包含主机列表。返回的数据格式是 JSON 格式。

❑ `--host <hostname>`：返回该主机的变量列表，或者是返回一个空的字典，使用 JSON 格式。

Ansible 使用 `-i` 选项来调用脚本。命令格式如下：

```
ansible all -i my-inventory-script -m ping
```

虽然在命令中并未体现，但 Ansible 默认是通过调用脚本的 `--list` 选项来获取 JSON 代码的。下面是一段由脚本生成的 JSON 代码。

```
{
  "databases": {
    "hosts": [
      "192.168.28.71",
      "192.168.28.72"
    ],
    "vars": {
      "ansible_ssh_user": "johndoe",
      "ansible_ssh_private_key_file": "~/.ssh/mykey",
      "example_variable": "value"
    }
  },
  "_meta": {
    "hostvars": {
      "192.168.28.71": {
        "host_specific_var": "bar"
      },
      "192.168.28.72": {
        "host_specific_var": "foo"
      }
    }
  }
}
```

在本例中，`databases` 为主机组名，可自定义。`hosts` 为固定字段，用于以列表形式定义主机组的主机。`vars` 也为固定字段，用于为主机组设置主机组变量。字典 `_meta` 中定义的是主机变量。

主机变量并不是 Inventory 文件中必需的，所以 `_meta` 字典也不是必须生成的。当 Inventory 脚本中生成 `_meta` 字典时，Ansible 会将 `_meta` 信息存放在缓存中，当任务中需要调用这些主机变量时，会直接从缓存中读取，而不是调用一次变量就执行一次 Inventory 脚本。这样就大大提高了运行效率。

1. 动态 Inventory 脚本的 Python 实现

我们使用 Vagrant 创建虚拟机，来演示动态 Inventory 脚本的基本写法。当然其他虚拟化平台的虚拟机或其他测试主机也都是可以的。首先在一个空目录下创建 Vagrant 的配置文件 Vagrantfile，内容如下：

```
VAGRANTFILE_API_VERSION = "2"
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.ssh.insert_key = false
  config.vm.provider :virtualbox do |vb|
    vb.customize ["modifyvm", :id, "--memory", "256"]
  end

  # Application server 1.
  config.vm.define "inventory1" do |inventory|
    inventory.vm.hostname = "inventory1.dev"
    inventory.vm.box = "geerlingguy/ubuntu1404"
    inventory.vm.network :private_network, ip: "192.168.28.71"
  end

  # Application server 2.
  config.vm.define "inventory2" do |inventory|
    inventory.vm.hostname = "inventory2.dev"
    inventory.vm.box = "geerlingguy/ubuntu1404"
    inventory.vm.network :private_network, ip: "192.168.28.72"
  end
end
```

然后运行 `vagrant up` 来启动这两台虚拟机，按照我们 Vagrantfile 中的配置，两台虚拟机都采用 Ubuntu 14.04 系统，IP 地址分别为 192.168.28.71 和 192.168.28.72。

要求最终产生的 JSON 代码需要等效于这样一份 Inventory 文件：

```
[group]
192.168.28.71 host_specific_var=foo
192.168.28.72 host_specific_var=bar

[group:vars]
ansible_ssh_user=vagrant
ansible_ssh_private_key_file=~/.vagrant.d/insecure_private_key
example_variable=value
```

下面我们来看一个最基本的基于 Python 开发的动态 Inventory 脚本，如代码清单 7-3 所示。

代码清单 7-3 Python 脚本

```

1 #!/usr/bin/env python
2
3 ...
4 基于 Python 的动态 Inventory 脚本举例
5 '''
6
7 import os
8 import sys
9 import argparse
10
11 try:
12     import json
13 except ImportError:
14     import simplejson as json
15
16 class ExampleInventory(object):
17
18     def __init__(self):
19         self.inventory = {}
20         self.read_cli_args()
21
22         # 定义 '--list' 选项
23         if self.args.list:
24             self.inventory = self.example_inventory()
25         # 定义 '--host [hostname]' 选项
26         elif self.args.host:
27
28             self.inventory = self.empty_inventory()
29         # 如果没有主机组或变量要设置, 就返回一个空 Inventory
30         else:
31             self.inventory = self.empty_inventory()
32
33         print json.dumps(self.inventory);
34
35     # 用于展示效果的 JSON 格式的 Inventory 文件内容
36     def example_inventory(self):
37         return {
38             'group': {
39                 'hosts': ['192.168.28.71', '192.168.28.72'],
40                 'vars': {
41                     'ansible_ssh_user': 'vagrant',
42                     'ansible_ssh_private_key_file':
43                         '~/.vagrant.d/insecure_private_key',
44                     'example_variable': 'value'
45                 }
46             },
47             '_meta': {
48                 'hostvars': {
49                     '192.168.28.71': {

```

```

50         'host_specific_var': 'foo'
51     },
52     '192.168.28.72': {
53         'host_specific_var': 'bar'
54     }
55 }
56
57 }
58
59 # 返回仅用于测试的空 Inventory
60 def empty_inventory(self):
61     return {'_meta': {'hostvars': {}}}
62
63 # 读取并分析读入的选项和参数
64 def read_cli_args(self):
65     parser = argparse.ArgumentParser()
66     parser.add_argument('--list', action = 'store_true')
67     parser.add_argument('--host', action = 'store')
68     self.args = parser.parse_args()
69
70 # 获取 Inventory
71 ExampleInventory()

```

将代码清单 7-3 保存为文件 `inventory.py`，并赋予其可执行权限（`chmod +x inventory.py`）。在命令行中使用其 `--list` 选项来测试其功能。

```
./inventory.py --list
```

运行结果如下：

```

{"group": {"hosts": ["192.168.28.71", "192.168.28.72"], "vars": {"ansible_
ssh_user": "vagrant", "ansible_ssh_private_key_file": "~/.vagrant.d/
insecure_private_key", "example_variable": "value"}}, "_meta": {"hostvars":
{"192.168.28.72": {"host_specific_var": "bar"}, "192.168.28.71": {"host_
specific_var": "foo"}}}}

```

使用 Ansible 命令调用这个脚本来测试两台虚拟机的网络是否正常。

```
ansible all -i inventory.py -m ping
```

运行结果如下：

```

192.168.28.71 | success >> {
"changed": false,
"ping": "pong"
}
192.168.28.72 | success >> {
"changed": false,
"ping": "pong"
}

```

确认通信没问题后，接下来验证我们设置的主机变量 `host_specific_var` 是否生效。命令如下：

```
ansible all -i inventory.py -m debug -a "var=host_specific_var"
```

运行结果如下：

```
192.168.28.71 | success >> {
  "var": {
    "host_specific_var": "foo"
  }
}
192.168.28.72 | success >> {
  "var": {
    "host_specific_var": "bar"
  }
}
```

由结果可以看出，主机变量的设置也是正确的。

若读者在实际生产中要用本例中的脚本，唯一要变动的是 `example_inventory()` 函数。为演示之用，写成了静态格式。在现实环境中，需要结合自身的业务场景编写代码，既可以通过调用外部 API 也可以查询数据库来取得所需的主机信息，并将其最终转换为 JSON 代码，供 Ansible 使用。

2. 动态 Inventory 脚本的 PHP 实现

我们可以使用任意编程语言来实现动态 Inventory 脚本。比如上一小节中基于 Python 实现的动态 Inventory 脚本，通过以下 PHP 代码也可以实现相同功能。

```
1 #!/usr/bin/php
2 <?php
3
4 /**
5  * @file
6  * 基于 PHP 的动态 Inventory 脚本举例
7  */
8
9 /**
10  *
11  *
12  * @return array
13  * 生成用于展示效果的 JSON 格式的 Inventory 文件内容
14  */
15 function example_inventory() {
16     return [
17         'group' => [
18             'hosts' => ['192.168.28.71', '192.168.28.72'],
19             'vars' => [
20                 'ansible_ssh_user' => 'vagrant',
```



```

21 'ansible_ssh_private_key_file' => '~/vagrant.d/insecure_private_key',
22 'example_variable' => 'value',
23 ],
24 ],
25 '_meta' => [
26 'hostvars' => [
27 '192.168.28.71' => [
28 'host_specific_var' => 'foo',
29 ],
30 '192.168.28.72' => [
31 'host_specific_var' => 'bar',
32 ],
33 ],
34 ],
35 ];
36 }
37
38 /**
39 * 生成用于测试的空 Inventory
40 *
41 * @return array
42 * 生成用于测试的空 Inventory
43 */
44 function empty_inventory() {
45     return ['_meta' => ['hostvars' => new stdClass()]];
46 }
47
48 /**
49 * 获取 Inventory
50 *
51 * @param array $argv
52 * 以数组形式传入变量 (as returned by $_SERVER['argv']).
53 *
54 * @return array
55 *
56 */
57 function get_inventory($argv = []) {
58     $inventory = new stdClass();
59
60     // 设置 '--list' 选项
61     if (!empty($argv[1]) && $argv[1] == '--list') {
62         $inventory = example_inventory();
63     }
64     // 定义 '--host [hostname]' 选项
65     elseif (!empty($argv[1]) && $argv[1] == '--host') && !empty($argv[2])) {
66         // 获取用于测试的空 Inventory
67         $inventory = empty_inventory();
68     }
69     // 如果没有主机组或变量要设置，就返回一个空 Inventory
70     else {

```

```

71 $inventory = empty_inventory();
72 }
73
74 print json_encode($inventory);
75 }
76
77 // 获取 Inventory
78 get_inventory($_SERVER['argv']);
79
80 ?>

```

将上述代码保存在文件 `inventory.php` 中，并赋予其可执行权限（`chmod +x inventory.php`）后，就可以像调用上面的 Python 脚本 `inventory.py` 一样，在 Ansible 中使用 `inventory.php` 了。

7.5 本章小结

通过本章的学习，我们可以看到，无论是简单到只有一台主机的架构还是主机数量上千台的服务器架构，借助 Ansible 的动态 Inventory 系统，我们可以轻松地实现对主机架构的动态管理，为快速扩张的公司业务提供强有力的服务器支持。

Ansible 插件扩展

截至本章，Ansible 所有的技术知识点均已介绍完毕，Ad_Hoc、Playbook、Inventory、Jinja、Galaxy、Roles 等技术点在前面章节从基础到入门篇到高级进阶篇均有详细介绍，这些内容已可满足日常工作中各类业务所需。但对于一些自定义或者特殊个性化需求还无法满足，Ansible 插件扩展功能提供高度自定义的扩展接口，读者具备代码基础即可编写属于自己的扩展插件，但是很少用户有这类需求，但为了书籍内容的完整性，本章也会介绍如何编写 Ansible 的插件扩展，但多数只是点到为止，推荐具备编码基础的读者深入学习第 12 ~ 14 章，自行开发 Web 界面。

8.1 Ansible 插件使用场景

如前文绍，Ansible 核心功能及自身 500 多个功能模块已能满足企业绝大多数场景所需。那么究竟什么时候我们需要自己开发插件来满足特有需求呢？这里不能一一涉及，大概列举如下场景供大家参考：

- 1) 除 Paramiko、本机 SSH、Local、Winrm 连接方式外，希望 Ansible 基于新的通信方式与远程主机交互；
- 2) 除 Ansible 内置的 with_items、with_fileglob 循环体外，希望有新的遍历方式；
- 3) 除了 Ansible 内置的 host_vars、group_vars 等变量调用方式外，希望有新的变量定义方式；
- 4) 除了 Ansible 的内置的 Jinja2 模板渲染、to_yaml、to_json 等过滤器外，希望有新的

过滤器；

5) 定义新的回调机制，即捕获响应事件后自定义新的响应形式。

□ 丰富强化标准的 Stdout 输出结果；

□ 增加日志记录的行为方式，如插入 MySQL、Redis、MongoDB 数据库；

□ 增加事件响应方式，如 Playbook 执行结果为 Success 时发送邮件给各部门组织新一轮的测试工作，免去人工干预过程；Playbook 执行结果为 Failed 时发送短信到自己手机上，尽快人为干预排查错误等。

如上场景需自行编写 Ansible 插件方可实现，类似场景可采用自行编写插件的方式满足。

8.2 Ansible 插件类型

Ansible 官方代码更新速度非常快，以致于官网的更新维护速度赶不上代码的更新速度。1.9 版本的插件类型支持 callback、connection、lookup、vars、filter、inventory 共 6 种类型的插件。从 2.0 版本开始，支持的插件类型几乎翻了一倍，又增加了 cache、inventory、shell、strategy、test 这 5 种类型，达到 11 种类型。但是在最新的 Devel 版本中，又将 inventory 插件功能取消了，所以大家在升级新版本时也需多关注其 Changelog 变更信息。我们将挑选一些大家可能涉及的插件做介绍，着重介绍 callback、connection、lookup、vars、filter，其他 5 种类型大家可参考 GitHub 网站^①自行研究。下面为大家介绍日常工作中可能用到的插件。

(1) Connection 类型插件

这类插件代表通信连接，用来和远程主机通信。默认提供 Paramiko、Native SSH、Local、Winrm 等连接方式。默认通过 ansible_connections 变量指定连接类型，如 `ansible all -m ping --connections=ssh`，或直接在 Inventory 文件中配置 `ansible_connection=winrm`，默认是系统自动判断连接类型，默认值是 smart。将新的 Connection 插件放在 `ansible.cfg` 指定的同级目录下即可生效。

(2) Lookup 类型插件

这类插件代表循环体功能类型，实现诸如 with_items、with_fileglob 等遍历功能的内置插件，Ansible-playbook 中的 with_items、with_fileglob 语法均是通过调用 Lookup 插件实现的。新的 Lookup 插件放在 `ansible.cfg` 指定的同级目录下即可生效。

(3) Vars 类型插件

从名称看可以得知是变量类型插件，这些变量并非来自 Inventory、Playbook、命令终端，而是通过 host_vars、group_vars 产生的，需要留意的是这些变量也可以通过 Inventory 产生，言外之意是 Vars 类型的插件绝大多数时候用不到。

① Plugins 网址：<https://github.com/ansible/ansible/tree/stable-2.1/lib/ansible/plugins>。

(4) Filter 类型插件

Filter 类型插件其实是 Jinja2 模板引擎的 Filter，Jinja2 的常用 Filter 实现有 `to_yaml`、`to_json`，官网实现的 Filter Plugins 代码全合并并在 `core.py`^①脚本中，新 Filter 插件需在该脚本的基础上编写。

(5) Callback 类型插件

Callback 类型插件是本章着重介绍的内容，该插件允许程序捕获响应的事件，并进行一些自定义响应，如前面提到的插入日志到数据库、发送邮件等功能。该类功能对运维的日常工作还是有些帮助的，因为我们可以通过统计数据库对平时的发布频次、成功 / 失败率、代码质量进行大数据处理和可预见性分析。

官方也提供了很多 Callback 类型插件，如 `log_plays`^②捕获 Playbook 事件日志写入文件，并在 Playbook 执行完毕时发送邮件给相关人员；`syslog_json`^③将 Ansible 日志以 JSON 格式输出等。同时 Ansible 还支持通过命名的方式按序执行插件，如希望插件第一个执行，则命名为 `l_first.py`，如希望最后一个执行，则命名为 `z_last.py` 即可。下面我们学习如何编写自己的插件。

8.3 如何编写自己的插件

首先大家要带着轻松的心态看待自定义插件，因为官方提供了很规范的代码模板供我们参考，新插件的编写其实就是在模板的基础上进行改写而已，所以不必有过重的心理负担。默认情况下插件案例放在当前系统环境 Python 安装路径下的 `site-packages/ansible/plugins/` 目录下，如 `/usr/lib/python2.6/site-packages/ansible/plugins/`。

其次编辑 `ansible.cfg` 定义插件存放目录，默认配置如下。

`ansible.cfg` 中在默认定义的插件目录下编写自己的插件。

```
action_plugins      = /usr/share/ansible_plugins/action_plugins
callback_plugins    = /usr/share/ansible_plugins/callback_plugins
connection_plugins  = /usr/share/ansible_plugins/connection_plugins
lookup_plugins      = /usr/share/ansible_plugins/lookup_plugins
vars_plugins        = /usr/share/ansible_plugins/vars_plugins
filter_plugins      = /usr/share/ansible_plugins/filter_plugins
strategy_plugins    = /usr/share/ansible_plugins/strategy_plugins
```

最后，从 GitHub 下载对应类型的模板到对应的目录修改即可，在模板的基础上修改即可。下面我们看案例。

① `core.py` 脚本网址：<https://github.com/ansible/ansible/blob/stable-2.1/lib/ansible/plugins/filter/core.py>。

② `log_plays` 脚本网址：https://github.com/ansible/ansible/blob/devel/lib/ansible/plugins/callback/log_plays.py。

③ `syslog_json` 脚本网址：https://github.com/ansible/ansible/blob/devel/lib/ansible/plugins/callback/syslog_json.py。

8.4 插件案例实践

对于插件这部分的应用，目前官方网站介绍的也不是很详细，而且不使用插件一样可以完成我们的所有工作，但插件的好处在于在编写 YAML 文件时可以减少我们的工作量，而且结果易于展示，所以我们只要学习一些对我们来说比较重要的比如 Filter、Callbacks 等即可。

对于 lookup_plugins 插件的使用，笔者不做细讲，日常用到的情况不是很多，我们可以作为读取文件内容的插件来使用。例如读取文件 filename 的内容：

```
set_fact:
  data: "{{ lookup('file', 'filename') }}"
```

上述 yaml 内容，我们只要把 filename 替换成自己的文件就可以完成读取文件内容的功能了。

我们着重来讲 filter 插件的使用，它将在我们编写 YAML 文件时，介绍我们的工作量。我们把过滤方法都提取出来写成公共的方法。

在普通情况下，我们主要是以 {{somevars | filter}} 对 somevars 使用 filter 方法过滤，Ansible 已经为我们提供了很多的过滤方法，比如找到列表中最大、最小数的 max、min，把数据转换成 JSON 格式的 fromjson 等，但这还远远不够，我们还需要自定义一些过滤的方法，来满足一些特殊的需求，比如查找列表中大于某个常数的所有数字等。我们接下来就会以这个为简单的例子，为读者展示如何去编写。

首先，到 ansible.cfg 中去掉 #，打开 filter_plugins 的存放目录，filter_plugins= /usr/share/ansible/plugins/filter。

我们来编写 deal_list_num.py 文件，它主要提供过滤列表中的正数、负数和查询列表中大于某一个给定数值这 3 个方法。来看下 /usr/share/ansible/plugins/filter/deal_list_num.py。

```
class FilterModule(object):
    def filters(self):      # 把使用的方法写在这个 return 中
        filter_map = {
            'positive': self.positive,
            'negative': self.negative,
            'no_less_than': self.no_less_than
        }
        return filter_map
    def positive(self, data):
        r_data = []
        for item in data:
            if item >= 0:
                r_data.append(item)
        return r_data
    def negative(self, data):
        r_data = []
        for item in data:
            if item <= 0:
```

```

        r_data.append(item)
    return r_data
def no_less_than(self, data, num): # 第1个变量为传入的数值, 第2个为条件1
    r_data = []
    for item in data:
        if item >= num:
            r_data.append(item)
    return r_data

```

我们编写 `deal_list_num.yml` 来使用我们上面的 3 个方法: `positive`、`negative` 和 `no_less_than`。

`deal_list_num.yml` 文件的内容如下:

```

- hosts: localhost
  gather_facts: False

  tasks:
    - name: set fact
      set_fact:
        num_list: [-1, -2, 5, 3, 1]

    - name: echo positive
      shell: echo {{num_list| positive}}
      register: print_positive
    - debug: msg="{{print_positive.stdout_lines[0]}}"

    - name: echo negative
      shell: echo {{num_list| negative}}
      register: print_negative
    - debug: msg="{{print_negative.stdout_lines[0]}}"

    - name: echo negative
      shell: echo {{num_list| no_less_than(4)}}
      register: print_negative
    - debug: msg="{{print_negative.stdout_lines[0]}}"

```

运行 `ansible-playbook deal_list_num.yml` 后返回的结果如下:

```

PLAY [localhost] *****
TASK: [set fact] *****
ok: [localhost]
TASK: [echo positive] *****
changed: [localhost]
TASK: [debug msg="{{print_positive.stdout_lines[0]}}"] *****
ok: [localhost] => {
  "msg": "[5, 3, 1]"
}
TASK: [echo negative] *****
changed: [localhost]
TASK: [debug msg="{{print_negative.stdout_lines[0]}}"] *****

```

```

ok: [localhost] => {
    "msg": "[-1, -2]"
}
TASK: [echo negative] *****
changed: [localhost]
TASK: [debug msg="{{print_negative.stdout_lines[0]}}"] *****
ok: [localhost] => {
    "msg": "[5]"
}

```

另一个比较重要的插件就是 `callback` 插件，该部分内容比较重要，而且与后续代码开发结合比较紧密，所以我们将放在第 12 章结合例子进行详细介绍。其他插件我们一般用到的也比较少，也不做过多介绍了。

8.5 本章小结

本章内容介绍的比较简单，因为我们真正在实际应用中不会用到太多，但笔者还是希望大家掌握几个比较重要的插件，比如 `filter` 和 `callback` 插件，灵活应用这两款插件将会给我们编写 YML 带来极大的方便，我们可以以极少且极易看懂的 YML 语法来完成一些看上去比较复杂的功能。这可能需要一些 Python 的基础，但读者可以按照 8.4 节的内容来模仿着编写，相信也不会很难。

Ansible 企业应用实战

经过前 8 章对 Ansible 从最基础的发展史，到 Ad-Hoc、Inventory、YAML、Playbook、Roles 等中高级用法的学习实践，截止到本章，恭喜诸位终于开启了 Ansible 企业应用实战的学习之旅。在接下来的章节中，各位将相继接触为新系统添加安全认证 SSHKey 及 Ansible 与行业主流技术的组合，如 ELK、Zabbix、GFS、Docker、Git 等，但都不会特别深入地介绍到方方面面，因为每个公司的业务特色都大相径庭，相信经过前 8 章的内容及实战引导，现在的你应对起来早已经绰绰有余。而在第 12 ~ 14 章将为大家介绍 Ansible 的 Web 自动化，带大家实现最终的 Web 自动化。接下来，我们先开始本章的企业应用实战。

9.1 为新系统添加安全认证 SSHKey

关于这节有必须先解释下，相信“老司机”心中早有数种招式，精通 Shell 的 Expect 可一招制敌，懂 Python 的 Paramiko、Expect 以现成模块套用方便之极，Ruby 通过 Net-ssh 也是很方便，但对于新手来讲可就没有如此多套路了。当然，这其中的大部分招式在本节都会介绍到。其实，Ansible 自身也支持密码认证，只是 Ansible 基于 SSHKey 认证的机制容易让人忽略其密码认证的功能，加之官网对该功能的引导确实不到位，所以该功能并不“出名”，但稍经深挖还是能找到的。本节我们也将先为大家介绍该方案。

9.1.1 Ansible 密码认证

因为密码在 Inventory 是明文配置的，考虑到安全性等其他原因，该方式只建议在首次

添加 SSHKey 认证时使用, 其他时间建议使用 SSHKey 认证方式。我们具体看下该方式如何实现。

步骤 1: 配置 Inventory, 默认配置 /etc/ansible/hosts, 添加配置如下。

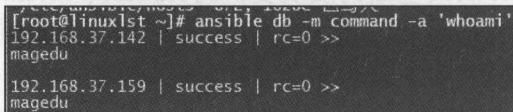
```
# 定义 db 组
[db]
192.168.37.142
192.168.37.159

# 冒号分隔, vars 定义变量, 改变 db 主机组的默认变量
[db:vars]
# 指定默认 ssh 用户为 magedu
ansible_ssh_user="magedu"
# 指定 ssh 用户密码
ansible_ssh_pass="magedu@beijing"
```

步骤 2: 测试默认变量是否生效, 执行以下命令。

```
ansible db -m command -a 'whoami'
```

查看当前用户的返回结果, 如图 9-1 所示为正常。



```
[root@linux1st ~]# ansible db -m command -a 'whoami'
192.168.37.142 | success | rc=0 >>
magedu
192.168.37.159 | success | rc=0 >>
magedu
```

图 9-1 查看当前用户

步骤 3: 调用 Ansible authorized_key 模块, 添加认证至远程主机。

```
ansible db -m authorized_key -a "user=magedu key='{{ lookup('file', '/home/magedu/.ssh/id_rsa.pub') }}' path=/home/magedu/.ssh/authorized_keys manage_dir=no"
```

或者调用 Ansible copy 和 shell 模块也可以实现。

```
# copy 公钥至远程主机 /tmp 目录下
ansible db -m copy -a "src=/home/magedu/.ssh/id_rsa.pub dest=/tmp/id_rsa.pub"
# 添加公钥
ansible db -m shell -a "cat /tmp/id_rsa.pub >> /home/magedu/.ssh/authorized_keys"
```

剩下就是验收了, 在 Master 机执行如下命令尝试是否能直接登录, 无密码登录即为正常。

```
ssh magedu@192.168.37.142
```

9.1.2 ssh-copy-id

命令 ssh-copy-id 用于复制指定用户的公钥至远程服务器, 同时修改 ~/.ssh 的目录权限。

Linux 入门必备，因其功能的局限性，主机较多时频繁输入密码较麻烦，主机量少时可以使用。具体方法如下。

命令使用方法：

```
/usr/bin/ssh-copy-id [-i [identity_file]] [user@]machine
```

为用户 magedu 生成认证（假设本地已存在公私钥）。

```
ssh-copy-id -i /home/magedu/.ssh/id_rsa.pub magedu@192.168.37.142
```

如上即已实现认证，如大量主机，请自行写批量认证脚本或参考本节其他方式。

9.1.3 Kickstart

试想，如果每个新系统都需添加认证，这是一件何等麻烦的事情。那为什么不让这些新系统“与生俱来”就拥有该认证呢？这是一个好问题，作为 RedHat 下主要的自动化安装配置工具的 Kickstart 可以帮我们完成该任务。完整的 Kickstart 可以通过 system-config-kickstart 图形界面生成，因非本书重点这里不做介绍，更多详细的内容可参考红帽官方 Kickstart 介绍^①。假设我们已经生成了一份 Kickstart 文件，那么在已有的配置添加如下代码即可：

```
...
/bin/cat <<EOF > /root/.ssh/authorized_keys
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAX4A5ghrljpBMldWpZlWC3gS6HWHluhOu5uDGjphXFC
YWTQ3ForfysVOMxJF2FHm1RAoxVr3zWQnduGQ1vEnWlWEmeFqEdyrgKUTCdZDQHdeeSPRka4MlGscM
aGtHQw0lW2Sbx6OSLcPrjimwVvJB54Fo6nUpta/cPrjY4P8bHN/CfMWxG5qHVG/LEPc6F8INS8006
QQ4sb5Stu4EnEgY5WyIMFL9PGCCVCzMjB8gTe6FGqP3SoD5k9Q5RAmWNqn+rgoGGjzTixWl5v7UVVt
RqGeolSPAniFDhkR0mKSQNFVTsaemp4nHvXmK9o89UnQs6k+CkzkzmsLRwppYw0XY5i8Q== magedu@
linuxlst
EOF
/bin/chmod 700 /root/.ssh/authorized_keys
%end
```

如上代码帮我们在新系统安装的时候内置了一份证书给 ROOT 用户，当然也可以指定认证任意用户。另外考虑到该功能的复杂度，建议同时添加主备 Master 管理机的认证，以防出现主 Master 管理机意外宕机造成大面积认证失败一时无法恢复的情况，同时私钥也建议加密后异地保存。在实现阶段，不少公司搭建自己的私有云 OpenStack、VMware vSphere 等，或已使用公有云 AWS、UCloud、阿里云、腾讯云等。更常用的技术是镜像克隆，所谓的镜像克隆可简单理解为将安装好的系统作为模板，有需要时拿来复制一份新系统即可，这些系统镜像也往往会经过一些简单的改造，如初始化目录、用户、认证，甚至是进程，根据业务模块生成不同的自定义系统镜像，最终实现工作量的最小化和质量的最可靠性。

① 红帽官方 Kickstart 介绍网址：https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Installation_Guide/s1-kickstart2-options.html。

9.1.4 Python Paramiko

Paramiko 是用 Python 语言写的一个模块，遵循 SSH2 协议，支持以加密和认证的方式进行远程服务器的连接。Paramiko 支持 Linux、Solaris、BSD、MacOS X、Windows 等平台通过 SSH 从一个平台连接到另外一个平台。利用该模块，可以方便地进行 SSH 连接，以及通过 SFTP 协议进行 SFTP 文件传输。

我们通过如下 `sshclient.py` 获取 192.168.37.142 主机名和 IP 信息的脚本来了解其使用。

```
# !/usr/bin/python
# -*- coding: utf-8 -*-
import paramiko

def sshe(ip, username, passwd, cmd):
    try:
        ssh = paramiko.SSHClient()
        ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
        ssh.connect(ip, 22, username, passwd)
        stdin, stdout, stderr = ssh.exec_command(cmd)
        print stdout.read()
        print "%s\tOK\n"%(ip)
        ssh.close()
    except :
        print "%s\tError\n"%(ip)
```

```
sshe("192.168.37.142", "linuxlst", "redhat", "hostname;ifconfig")
```

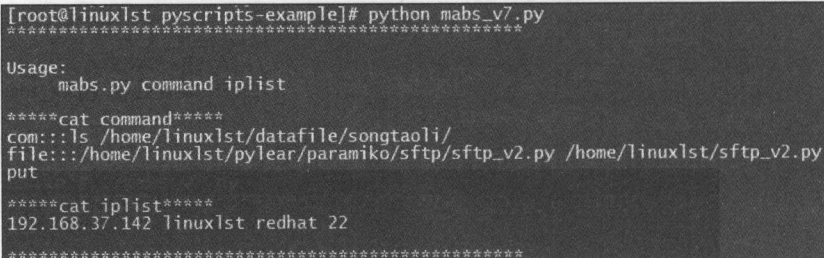
输出结果类似如下为正常：

```
linuxlst
```

```
192.168.37.142 OK
```

SSHClient 方法下的 `connect` 提供很多功能参数，本案例使用了 `ip`、`port`、`username`、`passwd`。另外也为大家准备了一个批量执行命令的 Python 脚本，请自行拉取^①。

不加任何参数直接运行会出现如图 9-2 所示的用法说明。



```
[root@linuxlst pyscripts-example]# python mabs_v7.py
*****
Usage:
  mabs.py command iplist

****cat command****
com::ls /home/linuxlst/datafile/songtaoli/
file::/home/linuxlst/pylear/paramiko/sftp/sftp_v2.py /home/linuxlst/sftp_v2.py
put

****cat iplist****
192.168.37.142 linuxlst redhat 22

*****
```

图 9-2 mabs_v7.py 用法说明

① Git 网址：[git@github.com:stanleyst/ansibleUI.git](https://github.com:stanleyst/ansibleUI.git)。

9.1.5 Expect

Expect 是 UNIX 系统中用来进行自动化控制和测试的软件工具，作为 Tcl 脚本语言的一个扩展应用在交互式软件中，如 Telnet、FTP、Passwd、FSCK、Rlogin、TIP、SSH 等。该工具利用 UNIX 伪终端包装其子进程，允许任意程序通过终端接入进行自动化控制；也可利用 Tk 工具，将交互程序包装在 X11 的图形用户界面中。

Expect 有利用正则表达式进行模式匹配以及通用的编程功能，允许用简单的脚本智能地管理如下工具：Telnet、FTP 和 SSH（这些工具都缺少编程的功能），宏以及其他程序。Expect 脚本的出现使得这些老的软件工具有了新的功能和更多的灵活性。

Expect 方式相对古老，但对于不会 Python 等语言但有 Bash 经验的人士来说是不错的选择。通过如下 expect.sh 来了解 Expect 的使用。

```
# !/usr/bin/expect

# 设置要连接的远程主机 IP 信息
set IP      [ lindex $argv 0 ]
# 设置要连接的远程主机登录用户
set USER    [ lindex $argv 1 ]
# 设置要连接的远程主机登录用户的密码信息
set PASSWD  [ lindex $argv 2 ]
# 设置要执行的命令
set CMD      [ lindex $argv 3 ]

# spawn 是 expect 内部命令，开启 ssh 连接
spawn ssh $USER@$IP $CMD
# 判断上次执行结果
expect {
    # 如果有 yes 或 no 关键字
    "(yes/no)?" {
        # 则输入 yes
        send "yes\r"
        # 输入完 yes 后如果输出结果有：password： 关键字
        expect "password:"
        # 则输入密码文件
        send "$PASSWD\r"
    }
    # 如果上次输出结果有：password：，则输入密码
    "password:" {send "$PASSWD\r"}
    # 如果上次输出结果有：* to host，则退出
    "** to host" {exit 1}
}
expect eof
```

代码注解已通过注释的方式添加，执行方式如下：

```
./expect.sh 192.168.37.142 linuxlst redhat /sbin/ifconfig
```

它唯一不友好的地方是密码明文，因环境变量没有加载所以执行命令需添加绝对路径。

不过这个问题通过强化 expect.sh 脚本均可解决。笔者曾使用过一个 700 行的 Expect Shell 脚本，堪称经典中的经典，因版权问题这里不做呈现。

为新系统添加认证的办法何其多，如上只是抛砖引玉，接下来我们进一步学习企业高可用架构的 Ansible 应用。

9.2 企业高可用架构的 Ansible 应用

在企业实际应用中，工程师非常注重架构的冗余性和可扩展性，这也使得生产环境的架构往往比较复杂但又不失灵活，这就要求管理工具也能够满足这些条件。本节我们参考开源爱好者分享的案例来为大家介绍通过 Ansible 部署一套完整的企业高可用架构的方法。涉及的流行技术有：Varnish、Apache、PHP、Memcached、MySQL。本次案例的架构拓扑图如 9-3 所示。

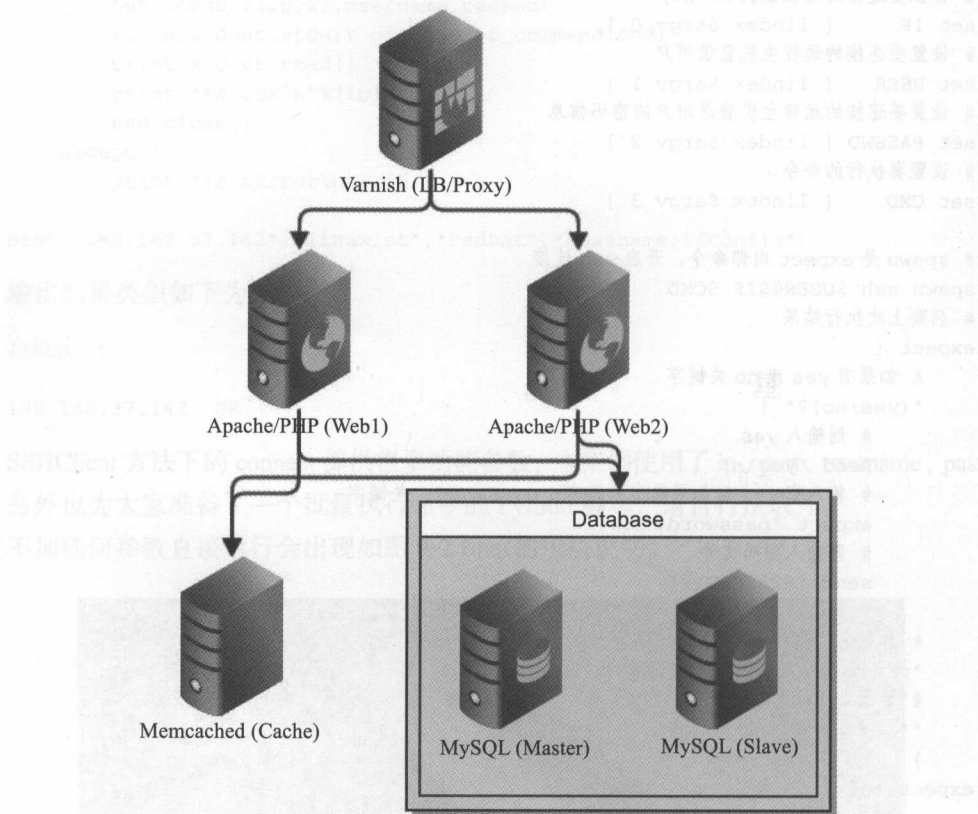


图 9-3 高可用架构图

□ Varnish 在前端接入层同时扮演负载均衡和反向代理的角色，将接收到的请求路由至

后面 WebServers。

- ❑ Apache 结合 PHP 的经典 WebServers 架构对外提供 Web 服务。
- ❑ 通过 Memcached 的 Cache 功能过滤数据库查询请求负载后，再将请求转发至后端 MySQL 数据库。
- ❑ MySQL 是通用主从架构，通常采用 InnoDB 存储引擎，主从时时同步，以主写从读的方式来缓解压力和提高存储层的可用性。

9.2.1 Playbook 目录编排

按照惯例，我们先来编排 Playbook 目录结构。

```
lamp-infrastructure/
  inventories/
  playbooks/
    db/
    memcached/
    varnish/
    www/
  provisioners/
  configure.yml
  provision.yml
  requirements.yml
  Vagrantfile
```

按这种目录编排的优势是方便我们聚焦单个服务器配置，在构建 Playbook 时可使用不同的 Inventory。另外，也可以使 Playbook 之间完全独立，模块化配置，提高功能复用率。

9.2.2 高可用架构基于 Ansible 的自动化实现

现在开始配置 Playbook。为提高效率，开发者已为我们提供了相关配置，我们只需从 Ansible Galaxy 下载下来即可，这些配置基于 CentOS 6.x 系统，Ubuntu、Debian 或者新版的 CentOS 稍做改动也可用。我们将基于 Varnish+Apache+PHP+Memcache+MySQL 来实现。下面我们依据图 9-3 高可用架构自上而下配置这些服务。

1. Varnish

编辑 playbooks/varnish/main.yml 文件，添加如下内容：

```
---
- hosts: lamp-varnish
  become: yes

  vars_files:
    - vars.yml # 变量定义文件

  roles:
```

```

- geerlingguy.firewall          # 配置防火墙，相关变量定义在 vars.yml 文件中
- geerlingguy.repo-epel         # 添加 EPEL 源
- geerlingguy.varnish           # 安装配置 Varnish

tasks:
- name: Copy Varnish default.vcl. # 通过 Jinja 模板生成 varnish 配置文件并传送到
                                  # 远程服务器
  template:
    src: "templates/default.vcl.j2"
    dest: "/etc/varnish/default.vcl"
    notify: restart varnish

```

该 YML 完成配置防火墙，添加 EPEL 库后安装配置 Varnish，并通过 Jinja 生成 Varnish 的 default.vcl 配置文件，完成负载均衡配置和其反向代理的功能。该 YML 定义调用的 vars.yml 是接下来我们需要定义的内容。在 main.yml 同级目录下编辑 vars.yml 文件，内容如下：

```

---
firewall_allowed_tcp_ports:
  - "22"
  - "80"

varnish_use_default_vcl: false

```

第 1 条变量通过 geerlingguy.firewall role 打开 IPTABLES TCP 协议的 22 和 80 的入链权限。第 2 条变量通过 geerlingguy.varnish role 配置 Varnish 使用自定义 default.vcl 配置。接着配置 default.vcl.j2 模板文件，通过 Jinja 模板转换可以帮我们完成 default.vcl 的配置，在 playbooks/varnish/templates/ 目录下编辑 default.vcl.j2 模板，内容如下：

```

1 vcl 4.0;
2
3 import directors;
4
5 {% for host in groups['lamp-www'] %}
6 backend www{{ loop.index }} {
7   .host = "{{ host }}";
8   .port = "80";
9 }
10 {% endfor %}
11
12 sub vcl_init {
13   new vdir = directors.random();
14   {% for host in groups['lamp-www'] %}
15     vdir.add_backend(www{{ loop.index }}, 1);
16   {% endfor %}
17 }
18
19 sub vcl_recv {

```



```

20 set req.backend_hint = vdir.backend();
21
22 # For testing ONLY; makes sure load balancing is working correctly.
23 return (pass);
24 }

```

我们这里不深入讲解 Varnish 的 VCL 语法，关于 Jinja 配置的部分这里简单说明如下。

- 第 1 ~ 3 行声明我们在使用 VCL 4.0 版本，Import Varnish directors 模块。
- 第 5 ~ 10 行定义后端的 Web Server 及端口号。
- 第 12 ~ 17 行定义 vcl_init，当 Varnish 启动和初始化任何模块时 vcl_init 即被调用。这里，我们定义负载均衡器 vdir，前面定义的 www backends 服务器为后端 Web Server，使用 random 调度方式。
- 第 19 ~ 24 行 vcl_recv 被调度，响应所有 Varnish 接收的请求至 vdir 定义的 backend 服务器。

该案例中我们有意调整 Varnish，只使用其负载均衡的功能，这是为了更方便地验证 Varnish 的请求调度流向，所以在实际生产应用中，请删除最后的 return (pass)，优化为业务实际所需后再加以使用，毕竟 Varnish 的卖点在于缓存机制。

2. Apache/php

在 playbooks/www/ 目录下，编辑 main.yml 内容如下：

```

---
- hosts: lamp-www
  become: yes

  vars_files:
    - vars.yml

  roles:
    - geerlingguy.firewall
    - geerlingguy.repo-epel
    - geerlingguy.apache
    - geerlingguy.php
    - geerlingguy.php-mysql
    - geerlingguy.php-memcached

  tasks:
    - name: Remove the Apache test page.
      file:
        path: /var/www/html/index.html
        state: absent

    - name: Copy our fancy server-specific home page.
      template:
        src: templates/index.php.j2
        dest: /var/www/html/index.php

```

和 Varnish 一样，Apache/PHP 也需要配置防火墙和 EPEL 源，部分代码解释如下。

- ❑ geerlingguy.apache: 安装配置 Apache Server。
- ❑ geerlingguy.php: 安装配置 PHP。
- ❑ geerlingguy.php-mysql: 添加 PHP 的 MySQL 插件支持。
- ❑ geerlingguy.php-memcached: 添加 PHP 的 Memcached 插件支持。

最后的两条 Tasks 命令删除 Apache 默认 index.html 配置，替换为 index.php。在该 YAML 文件同级目录下，我们编辑其调用的变量配置文件 vars.yml 内容如下：

```
---
firewall_allowed_tcp_ports:
  - "22"
  - "80"
```

index.php.j2 模板文件位于 playbooks/www/templates/ 目录下，该文件使用 Jinja 格式创建 PHP 文件，来展现服务器的健康状态。其内容如下：

```
1 <?php
2 /**
3  * @file
4  * 搭过测试页面
5  *
6  * 不要在生产环境中使用这段代码，它只是个简单的原型
7  */
8
9 $mysql_servers = array(
10 {% for host in groups['lamp-db'] %}
11   '{{ host }}',
12 {% endfor %}
13 );
14 $mysql_results = array();
15 foreach ($mysql_servers as $host) {
16   if ($result = mysql_test_connection($host)) {
17     $mysql_results[$host] = '<span style="color: green;">PASS</span>';
18     $mysql_results[$host] .= ' (' . $result['status'] . ')';
19   }
20   else {
21     $mysql_results[$host] = '<span style="color: red;">FAIL</span>';
22   }
23 }
24
25 // 连接 Memcached
26 $memcached_result = '<span style="color: red;">FAIL</span>';
27 if (class_exists('Memcached')) {
28   $memcached = new Memcached;
29   $memcached->addServer('{{ groups['lamp-memcached'][0] }}', 11211);
30
31   // 测试为 memcached 添加新变量
```

```

32 if ($memcached->add('test', 'success', 1)) {
33     $result = $memcached->get('test');
34     if ($result == 'success') {
35         $memcached_result = '<span style="color: green;">PASS</span>';
36         $memcached->delete('test');
37     }
38 }
39 }
40
41 /**
42  * 连接到 MySQL 服务器并测试连接
43  *
44  * @param string $host
45  * 服务器的 IP 地址或主机名
46  *
47  * @return array
48  * 'success' (bool) 和 'status' ('slave' or 'master') 数组
49  * 连接失败返回空
50 */
51 function mysql_test_connection($host) {
52     $username = 'mycompany_user';
53     $password = 'secret';
54     try {
55         $db = new PDO(
56             'mysql:host=' . $host . ';dbname=mycompany_database',
57             $username,
58             $password,
59             array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
60
61         // 询问服务器被配置的状态 master 或 slave
62         $statement = $db->prepare("SELECT variable_value
63             FROM information_schema.global_variables
64             WHERE variable_name = 'LOG_BIN'");
65         $statement->execute();
66         $result = $statement->fetch();
67
68         return array(
69             'success' => TRUE,
70             'status' => ($result[0] == 'ON') ? 'master' : 'slave',
71         );
72     }
73     catch (PDOException $e) {
74         return array();
75     }
76 }
77 ?>
78 <!DOCTYPE html>
79 <html>
80 <head>
81     <title>Host {{ inventory_hostname }}</title>

```

```

82 <style>* { font-family: Helvetica, Arial, sans-serif }</style>
83 </head>
84 <body>
85 <h1>Host {{ inventory_hostname }}</h1>
86 <?php foreach ($mysql_results as $host => $result): ?>
87 <p>MySQL Connection (<?php print $host; ?>):
88 <?php print $result; ?></p>
89 <?php endforeach; ?>
90 <p>Memcached Connection: <?php print $memcached_result; ?></p>
91 </body>
92 </html>

```

这些代码大家都可通过 [ansibleUI](https://github.com/stanleyst/ansibleUI)^① 下载，我们来快速浏览下该配置做了什么事情：

- 第 9 ~ 23 行遍历 Inventory 定义的 lamp-db 所有 MySQL 主机，测试其连接性。
- 第 25 ~ 39 行测试 Inventory 定义的 lamp-memcached 所有 Memcached 主机 create、retrieve、delete 权限是否正常
- 第 41 ~ 76 行定义 `mysql_test_connection()` 函数测试 MySQL Server 状态，判断主从关系。
- 第 78 ~ 91 行生成 MySQL、Memcached 的测试结果，并通过一个简易的 Web 页面展示出来。

3. Memcached

相对于 Varnish、PHP，配置 Memcached 要简单很多，创建 `playbooks/memcached/main.`

`yaml` 配置文件即可。内容如下：

```

1 ---
2 - hosts: lamp-memcached
3   become: yes
4
5   vars_files:
6     - vars.yml
7
8   roles:
9     - geerlingguy.firewall
10    - geerlingguy.memcached

```

该 YML 只有两个 Roles：一个开放防火墙确认端口可通；一个配置 memcached 的 Roles。

配置其变量文件 `vars.yml` 内容如下：

```

---
firewall_allowed_tcp_ports:
  - "22"
firewall_additional_rules:
  - "iptables -A INPUT -p tcp --dport 11211 -s {{ groups['lamp-www'][0] }} -j

```

① [ansibleUI](https://github.com/stanleyst/ansibleUI) 网址：<https://github.com/stanleyst/ansibleUI>，其原始网址：<https://github.com/geerlingguy/ansible-for-devops>。


```
ACCEPT"
- "iptables -A INPUT -p tcp --dport 11211 -s {{ groups['lamp-www'][1] }} -j
ACCEPT"
```

```
memcached_listen_ip: "{{ groups['lamp-memcached'][0] }}"
```

远程服务器 22 端口必须开放，以供远程 SSH 连接所需。另外开放了 Memcached 的 11211 端口，供 Web Server 进程通信使用。考虑到安全因素，我们限制源 IP 为 lamp-www 组定义的主机。

4. MySQL

MySQL 的配置更为复杂，因为除了基本架构配置外，我们还需要为其配置每个用户信息及主从复制关系，同时希望 Playbook 组织架构足够灵活可配置。也因为业务需求，我们需要 MySQL 在不同的环境下都可以动态获取服务器的 IP 信息。

先来配置 playbooks/db/main.yml 文件。

```
1 ---
2 - hosts: lamp-db
3   become: yes
4
5   vars_files:
6     - vars.yml
7
8   pre_tasks:
9     - name: Create dynamic MySQL variables.
10       set_fact:
11         mysql_users:
12           - name: mycompany_user
13             host: "{{ groups['lamp-www'][0] }}"
14             password: secret
15             priv: " *.*:SELECT"
16           - name: mycompany_user
17             host: "{{ groups['lamp-www'][1] }}"
18             password: secret
19             priv: " *.*:SELECT"
20         mysql_replication_master: "{{ groups['a4d.lamp.db.1'][0] }}"
21
22   roles:
23     - geerlingguy.firewall
24     - geerlingguy.mysql
```

我们在 pre_tasks 中额外使用了 set_fact 来动态生成变量信息。set_fact 允许 Playbooks 在被调用时定义变量，即我们可以为所有服务器生成相关变量，即使新加入的服务器也可生效。这里我们创建了两个变量：

□ mysql_users 指定的用户列表被 geerlingguy.mysql 调用创建，同时所有的 DB 服务器上执行，即两台 lamp-www 服务器会拥有所有 DB 的读权限。

□ `mysql_replication_master` 被 `geerlingguy.mysql` 用来指定主从关系，并根据不同的角色进行不同的配置。

其他的常规变量我们依旧定义在 `playbooks/db/vars.yml` 文件中。

```
---
firewall_allowed_tcp_ports:
  - "22"
  - "3306"

mysql_replication_user: {name: 'replication', password: 'secret'}
mysql_databases:
  - name: mycompany_database
    collation: utf8_general_ci
    encoding: utf8
```

这里我们开放了 3306 端口，不做限制，根据权限最小化原则，建议只针对特定的机器开放端口。另外两个 MySQL 变量，`mysql_replication_user` 用于 M/S 之间数据同步，`mysql_databases` 用于指定 MySQL 创建时的字符集等基础配置信息。

各应用独立模块配置完毕后，我们就可以开始顶层 Playbook 的配置工作了。

9.2.3 使用 Includes 衔接各服务配置

好的 Playbooks 通过简单的 Includes 即可完成整体架构的配置工作。在项目的根目录（即 `playbooks` 的同级目录）下配置 `configure.yml`，内容如下：

```
---
- include: playbooks/varnish/main.yml
- include: playbooks/www/main.yml
- include: playbooks/db/main.yml
- include: playbooks/memcached/main.yml
```

到目前为止，如果你的服务器运行正常，`lamp-www`、`lamp-db` 等 Inventory 均已配置，那么现在运行命令 `ansible-playbook configure.yml`，就可以完成 HA 高可用架构的配置工作了。

但其实现有的 Playbooks 可以更加灵活强大，限于篇幅我们将所有的 Roles 已经上传 Ansible Galaxy，通过 `ansible-galaxy install -r requirements.yml` 即可下载安装 YML 中所有 Galaxy Roles。`requirements.yml` 内容如下：

```
---
- src: geerlingguy.firewall
- src: geerlingguy.repo-epel
- src: geerlingguy.varnish
- src: geerlingguy.apache
- src: geerlingguy.php
- src: geerlingguy.php-mysql
```

```
- src: geerlingguy.php-memcached
- src: geerlingguy.mysql
- src: geerlingguy.memcached
```

命令执行完毕后，Galaxy Roles 默认存放在 /etc/ansible/roles/。

9.3 ELK 日志系统基于 Ansible 的自动化实现

Web 应用、数据存储、架构高可用及可拓展性无疑是 IT 架构中最为核心关键的部分，随互联网的高速发展，架构不断演变至今，企业的架构越来越复杂化、多样化，对 IT 的技能要求也越来越高，更需要技术人员在错综复杂的问题中快速定位问题所在，这就要求有一款日志收集分析系统能帮助运维快速定位问题，ELK（Elasticsearch、Logstash、Kibana）就是众多工具中的佼佼者。本节为大家介绍 ELK 日志系统基于 Ansible 的自动化实现。

与前面章节介绍案例的方式有所不同，自第 9 章开始会认为读者具备 Ansible 基础，因涉及代码较多所以不逐行分析，所有代码请从 GitHub^①获取（将 ansibleUI/roles/ 目录下的所有代码复制至自定义的 ROLES 目录下即可），本书只为大家介绍构架及逻辑重点。本次介绍的 ELK 架构 ELK-Infrastructure 如图 9-4 所示。

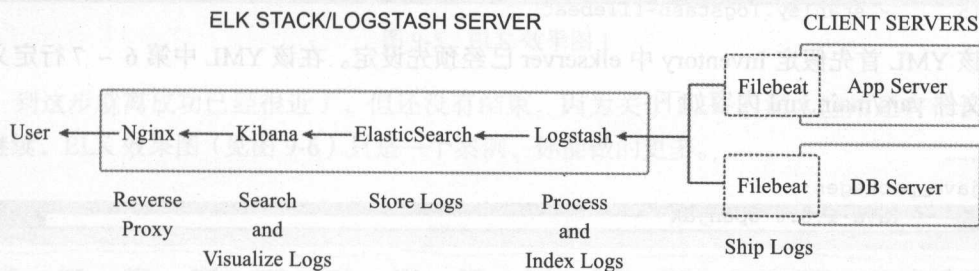


图 9-4 ELK-Infrastructure

- Filebeat Shipper：Logstash-forward 的替代版，作为 Client 安装在需收集日志的服务器上，采集日志并发送给 Logstash。
 - Logstash：二次处理收集的日志。
 - ElasticSearch：储存收集的日志。
 - Kibana：提供可视化的日志搜索查询平台。
- 该架构下需要 Ansible 完成的功能点如下：
- 安装 Java、ElasticSearch、Logstash、Kibana、Nginx、Filebeat 软件包。
 - CA 自签名认证。

① GitHub 网址：<https://github.com/stanleyst/ansibleUI.git>。

- ❑ 通过 `htpasswd` 为 Kibana 生成认证登录用户。
- ❑ 配置 ELK、Nginx、Filebeat 相互间配置信息。
- ❑ 加载 Kibana Dashboards 模板信息。

接下来我们先配置 Server 端，再配置 Client 端。

9.3.1 ELK Server 的自动化实现

先来看 ELK Server 的主配置文件 `ansibleUI/elk-example/elk.yml`。

```

1 ---
2
3 - hosts: elkserver
4   gather_facts: yes
5
6   vars_files:
7     - vars/main.yml
8
9   roles:
10    - geerlingguy.java
11    - geerlingguy.nginx
12    - geerlingguy.elasticsearch
13    - stanley.kibana
14    - geerlingguy.logstash
15    - stanley.logstash-filebeat

```

该 YML 首先假定 Inventory 中 `elkserver` 已经预先设定。在该 YML 中第 6 ~ 7 行定义的变量文件 `vars/main.yml` 内容如下：

```

---
java_packages:
  - java-1.8.0-openjdk

nginx_user: nginx
nginx_worker_connections: 1024
nginx_remove_default_vhost: true

kibana_server_name: logs
kibana_username: kibana
kibana_password: password
logstash_monitor_local_syslog: false

```

指定需安装的 Java 版本号、Nginx 模板中定义的变量、kibana 的登录用户信息等。

上述 YML 第 10 ~ 15 行引用 `geerlingguy.java`、`geerlingguy.nginx`、`geerlingguy.elasticsearch`、`stanley.kibana`、`geerlingguy.logstash`、`stanley.logstash-filebeat` Roles，分别配置 Java、Nginx、ElasticSearch、Kibana、Logstash、Logstash-filebeat 系统。

这时如果你已经迫不及待想看效果的话，那么直接执行 `ansible-playbook elk.yml` 就可以看到 ELK 的初步页面效果了。不过这条命令要执行的大概 30 分钟甚至更长时间，因为要不

停地下载安装。如果一切执行正常，类似如下绑定本机 logs.magedu.com 域名：

```
192.168.37.167 logs.magedu.com
```

浏览器显示 ELK 效果图如图 9-5 所示。

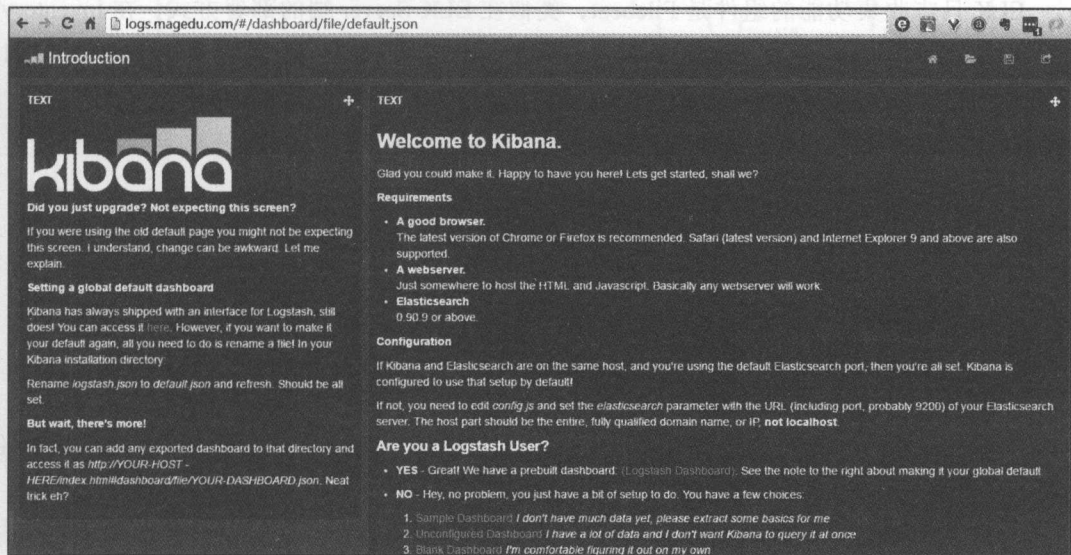


图 9-5 ELK 效果图 1

到这步就离成功已经很近了，但还没有结束，因为关于 ELK 的深入应用还有很长的路要继续，ELK 效果图（见图 9-6）只是一个案例，你能做的更多。

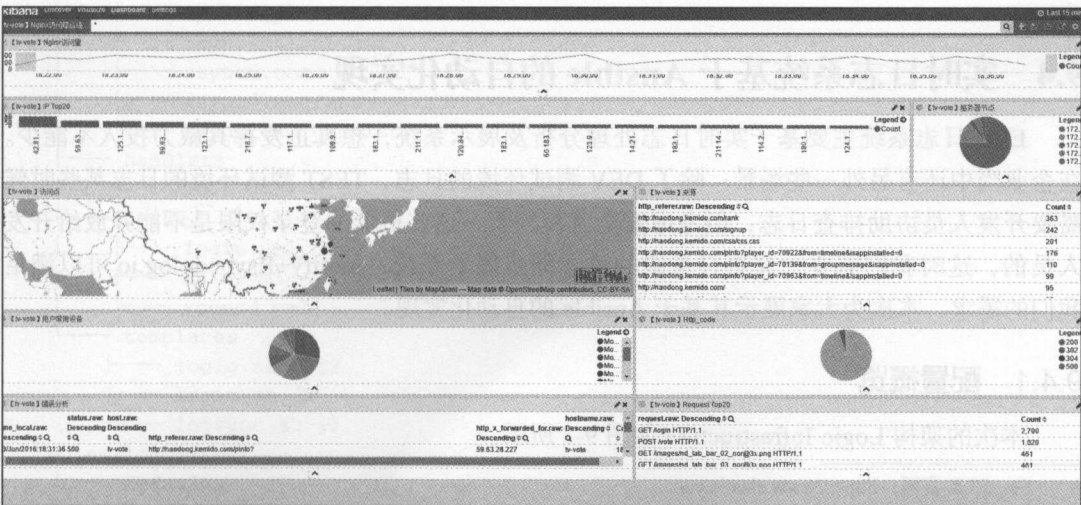


图 9-6 ELK 效果图 2

为了快速演示效果，我们将该案例所有的组件部署到一台服务器上，如果需要收集多台服务器的日志如何快速实现呢？

9.3.2 ELK Client 的自动化实现

ELK 日志收集功能的组件是 Filebeat，类似在 ELK Server 端的部署方式，在 Inventory 定义好 elkclient 后对其执行 stanley.logstash-filebeat 这个 ROLE 即可。整体 elk_filebeat.yml 内容如下：

```
---

- hosts: elkclient
  gather_facts: yes

  vars_files:
    - vars/main.yml

  roles:
    - stanley.logstash-filebeat
```

关于 ELK 的自动化部署代码介绍到这里结束。在如上部署中涉及安全性问题，大家需修改两点：

- ❑ CA 自签名证书目录位于 /etc/pki/logstash/，请务必使用自己的证书；
- ❑ 通过 htpasswd 为 Kibana 生成认证登录用户默认用户名 / 密码为：kibana/password，请修改后使用。

关于 ELK 的更深入学习请参考 ELK 官网^①，如上 ELK 所有代码请至 stanleylst 的 GitHub^② 下载源码。因 Logstash 软件包较大且官方下载较慢，网络不佳的朋友可从网盘地址^③下载。

9.4 实时日志系统基于 Ansible 的自动化实现

ELK 日志系统主要基于实时日志处理分析及展示系统，想真正发挥其威力投入不能少。在企业当中还有另外一些场景：除了 DEV 测试环境的日志，TEST 测试环境的日志某些时候需要开发人员协助排查日志，但考虑安全流程因素，这些机器的登录权限是不能开放给开发人员的，这时我们需要一套更轻量级的日志同步展示系统，Inotify+Rsyncd+log.io 可以满足我们的需求。本次为大家展示其基于 Ansible 的自动化实现。

9.4.1 配置概览

本次的架构 Logio Infrastructure 如图 9-7 所示。

① ELK 官网：<https://www.elastic.co/>。

② Stanleylst 的 GitHub 网址：<https://github.com/stanleylst/ansibleUI.git>。

③ Logstash 网盘网址：<http://pan.baidu.com/s/1eSLPQ8M>。

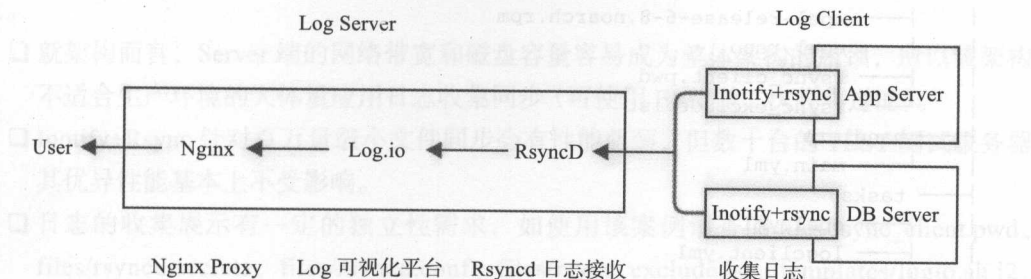


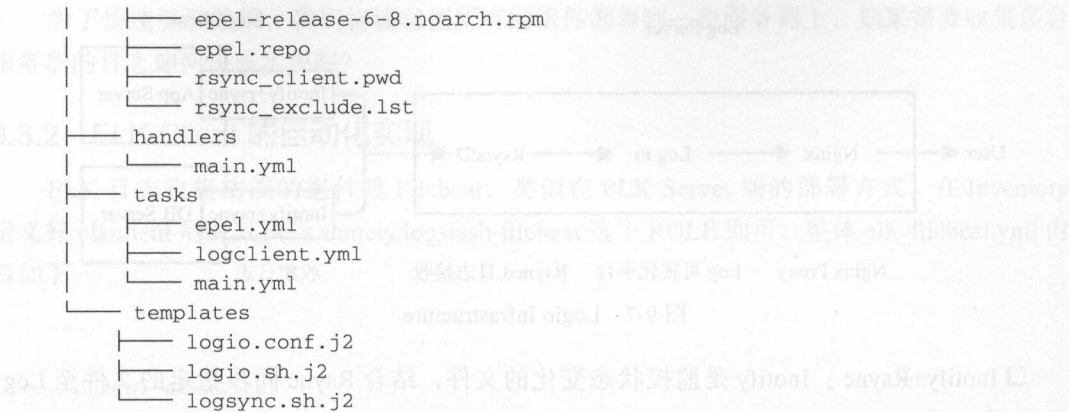
图 9-7 Logio Infrastructure

- Inotify+Rsync：Inotify 是监控状态变化的文件，结合 Rsync 同步指定的文件至 Log Server。
- RsyncD LogServer：开启 Rsync Daemon 服务，用于免密码交互文件接收。
- Log.io：提供日志展示平台。
- Nginx：反向代理请求至 Log.io，方便用户访问。

9.4.2 架构部署

Server 和 Client 端拆分不同的 Roles 方便部署，具体的目录结构如下：

```
log_client.yml
log_server.yml
roles/logsync/
├── defaults
│   └── main.yml
├── files
│   ├── epel-release-6-8.noarch.rpm
│   ├── epel.repo
│   ├── rsync_client.pwd
│   ├── rsyncd.conf
│   ├── rsyncd.secrets
│   └── rsync_exclude.lst
├── handlers
│   └── main.yml
├── tasks
│   ├── logio.yml
│   ├── main.yml
│   └── rsyncd.yml
└── templates
    ├── logio.conf.j2
    ├── logio.sh.j2
    └── logsync.sh.j2
roles/logclient/
├── defaults
│   └── main.yml
└── files
```



将 Roles 下代码下载至对应目录后可直接进行相关部署。Server 端部署方式如下：

```
ansible-playbook log_server.yml
```

Client 端部署方式如下：

```
ansible-playbook log_client.yml
```

其对应的 Logio 组件功能表如图 9-8 所示。

Roles		
log_server.yml	<div><div>tasks</div><div><div>main.yml</div><div>logio.yml</div><div>rsyncd.yml</div></div></div>	<div><div>1) Include 调用 logio.yml、rsyncd.yml 配置</div><div>1) Init Repo; 2) Install Npm, Nginx, logio; 3) Config logio;</div><div>1) 配置 Rsync Daemon</div></div>
log_client.yml	<div><div>tasks</div><div><div>main.yml</div><div>epel.yml</div><div>logclient.yml</div></div></div>	<div><div>1) Include 调用 logclient.yml、epel.yml 配置</div><div>2) Init Repo</div><div>3) 配置 Inotify, logio-client, Rsyncs 同步</div></div>

图 9-8 Logio 组件功能表

类似于本章其他案例，Server^①和 Client^②代码请从 GitHub 下载。该案例需要注意如下

① Logserver 代码：<https://github.com/stanleyst/ansibleUI/tree/master/roles/logsync>。
② Logclient 代码：<https://github.com/stanleyst/ansibleUI/tree/master/roles/logclient>。

几点。

- ❑ 就架构而言，Server 端的网络带宽和磁盘容量容易成为整体架构的瓶颈，所以该架构不适合生产环境的大体量应用日志收集同步（可使用 ELK 做二次收集过滤）。
- ❑ Inotify+Rsync 针对百万量级小文件同步会有性能瓶颈，但数十台的 TEST 测试服务器其优异性能基本上不受影响。
- ❑ 日志的收集展示有一定的独立性需求，如使用该案例请查阅 files/rsync_client.pwd、files/rsyncd.secrets、files/rsyncd.conf、files/rsync_exclude.lst、templates/logio.sh.j2、templates/logsync.sh.j2，并修改为自己所需后使用。

用户可优化该案例，并继续扩展其功能，如：Crontab 定期重启、日志切割及定期清理等。当然也能结合 Supervisor 就更好了。更多想法大家也可以在 GitHub 留言或扫描前言中的二维码关注后留言。

9.5 Zabbix 基于 Ansible 的自动化实现

Zabbix 是一个基于 Web 界面的、提供分布式系统监视以及网络监视功能的企业级开源解决方案，在企业级监控领域扮演日益重要的角色，官方网址为 <http://www.zabbix.com>，目前最新版本为 3.0。作为运维必备技能，本节为大家介绍如何通过 Ansible 实现 Zabbix-server、Zabbix-agent、zabbix-proxy 的批量部署配置。

Zabbix 在企业常用架构图如图 9-9 所示。

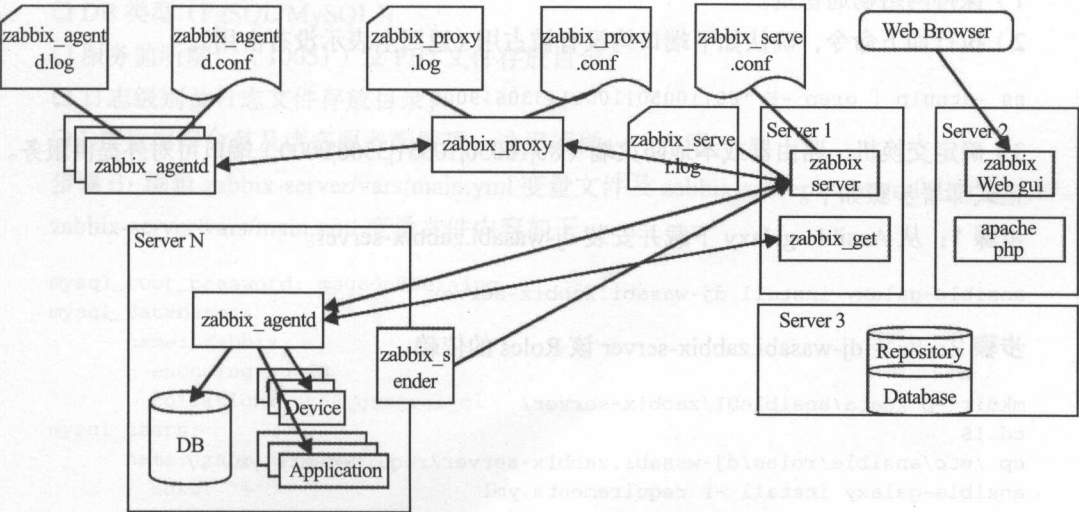


图 9-9 Zabbix 架构图

主要组件功能如下。

- ❑ Zabbix Server：负责接收 Agent 发送的报告信息的核心组件，所有配置、统计数据及

操作数据均由其组织进行;

- ❑ Database Storage: 专用于存储所有配置信息, 以及由 Zabbix 收集的数据;
- ❑ Web interface (frontend): Zabbix 的 GUI 接口, 通常与 Server 运行在同一台机器上;
- ❑ Zabbix Proxy: 可选组件, 常用于分布式监控环境中, 代理 Server 收集部分被监控数据并统一发往 Server 端;
- ❑ Zabbix Agent: 部署在被监控主机上, 负责收集本地数据并发往 Server 端或者 Proxy 端。

本节将为大家依次介绍 Zabbix Server、Zabbix Agent、Zabbix Proxy 基于 Ansible 的批量部署配置实现。

9.5.1 Zabbix Server 基于 Ansible 的自动化实现

我们选择 Ansible Galaxy 的 `dj-wasabi.zabbix-server` 完成 Zabbix Server 的安裝配置工作。该 Roles 支持 Debian、EL、Ubuntu 系统, 支持的 Zabbix 2.2、2.4、3.0 版本, 支持的 DB 类型有 MySQL、PgSQL。本次我们要部署的项目配置如下:

```
系统: Centos 6.8 64bit
Apache: 2.4.6
PHP: 5.6.23
MySQL: 5.1.73
Zabbix_agentd: 3.0.3
Zabbix_server: 3.0.3
```

开始前准备工作如下:

- 1) 保持网络畅通稳定。
- 2) 执行如下命令, 确认如下端口均没有被占用(返回空表示没有占用)。

```
ss -atnulp | grep -E '80|10050|10051|3306|9000'
```

- 3) 确定交换机、路由器或本地防火墙(80|10050|10051|3306|9000)端口可对外提供服务。正式部署步骤如下。

步骤 1: 从 Ansible-galaxy 下载并安装 `dj-wasabi.zabbix-server`。

```
ansible-galaxy install dj-wasabi.zabbix-server
```

步骤 2: 安装 `dj-wasabi.zabbix-server` 该 Roles 的依赖。

```
mkdir -p /data/ansibleUI/zabbix-server/
cd !$
cp /etc/ansible/roles/dj-wasabi.zabbix-server/requirements.yml ./
ansible-galaxy install -r requirements.yml
```

步骤 3: 设置系统 SELINUX 权限为 `permissive` 或 `disabled`, 不然启动服务会有权限问题。

```
# 临时生效
setenforce 0
```

```
# 永久生效
```

```
sed -i 's/SELINUX=enforcing/SELINUX=disabled/' /etc/selinux/config
```

步骤4：编辑 Inventory 文件 (/etc/ansible/hosts)，配置 zabbix-server。添加如下内容：

```
[zabbix-server]
192.168.37.167
```

步骤5：配置 zabbix-server 基础服务信息，修改配置文件 dj-wasabi.zabbix-server/defaults/main.yml，其他项如各自所需配置。建议修改项如下：

```
zabbix_url: zabbix.magedu.com
zabbix_url_aliases: []
zabbix_version: 3.0
zabbix_timezone: Asia/Shanghai
```

```
server_dbhost: localhost
server_dbname: zabbix-server
server_dbschema:
server_dbuser: zabbix-server
server_dbpassword: zabbix-server
```

dj-wasabi.zabbix-server/defaults/main.yml 的文件共 92 行，该文件主要定义了以下内容：

- ☐ Zabbix 服务登录域名；
- ☐ 服务时区；
- ☐ 是否开户 web、vhost、repo；
- ☐ DB 类型 (PgSQL/MySQL)；
- ☐ 服务监听端口 (10051) 及 PID 文件存放目录；
- ☐ 日志级别及日志文件存放目录；
- ☐ DB 数据库命名及诸多服务配置项，这里不做一一介绍。

步骤6：编辑 zabbix-server/vars/main.yml 变量文件及 zabbix-server/main.yml 主配置文件。

zabbix-server/vars/main.yml 变量文件内容如下：

```
mysql_root_password: magedu@beijing
mysql_databases:
  - name: zabbix
    encoding: utf8
    collation: utf8_general_ci
mysql_users:
  - name: zabbix-server
    host: "%"
    password: zabbix-server
    priv: "example_db.*:ALL"
```

zabbix-server/main.yml 主配置文件内容如下：

```
---
```

```

- hosts: zabbix-server
  vars_files:
    - vars/main.yml
  tasks:
    - name: Create Repo
      copy: src={{ item }} dest=/etc/yum.repo.d/{{ item }}
      with_items:
        - epel-httpd24.repo
        - remi-php70.repo
        - remi.repo
        - remi-safe.repo

    - name: Install PHP56
      yum: name={{ item }} state=present
      with_items:
        - php56.x86_64
        - php56-php-bcmath.x86_64
        - php56-php-cli.x86_64
        - php56-php-common.x86_64
        - php56-php-devel.x86_64
        - php56-php-embedded.x86_64
        - php56-php-fpm.x86_64
        - php56-php-gd.x86_64
        - php56-php-mbstring.x86_64
        - php56-php-mysqlnd.x86_64
        - php56-php-odbc.x86_64
        - php56-php-openssl.x86_64
        - php56-php-pdo.x86_64
        - php56-php-pear.noarch
        - php56-php-pecl-jsonc.x86_64
        - php56-php-pecl-jsonc-devel.x86_64
        - php56-php-pecl-zip.x86_64
        - php56-php-pgsql.x86_64
        - php56-php-process.x86_64
        - php56-php-xml.x86_64
        - php56-runtime.x86_64

    - name: Install Apache24
      yum: name={{ item }} state=present
      with_items:
        - httpd24
        - httpd24-httpd
        - httpd24-httpd-tools

- hosts: zabbix-server
  become: yes
  vars_files:
    - vars/main.yml
  roles:

```



```

- { role: geerlingguy.mysql }

- hosts: zabbix-server
  sudo: yes
  roles:
    - { role: geerlingguy.apache }
    - { role: dj-wasabi.zabbix-server, zabbix_url: zabbix.magedu.com, database_
      type: mysql, database_type_long: mysql }

```

该 YAML 没有参考 dj-wasabi 的 zabbix-server 官方安装，而是加入了 Apache2.4 和 PHP5.6 的安装工作，因为我们希望使用 Apache 最新的 php-fpm 功能。因为改动较大，所以该节代码建议从 GitHub 下载完整的代码。

步骤 7：部署安装 Zabbix-Server。

```
ansible-playbook main.yml
```

到此，所有部署工作结束，部署过程因需下载软件包，因此部署时长根据网络状况长短不一，期间请保证网络稳定畅通。安装结束后打开浏览器访问 <http://zabbix.magedu.com/index.php> (需事先绑定本机 HOSTS，HOSTS 文件添加内容 192.168.37.167 zabbix.magedu.com)，输入预先设定的用户名、密码后，会看到如图 9-10 所示的 Zabbix Dashboard。

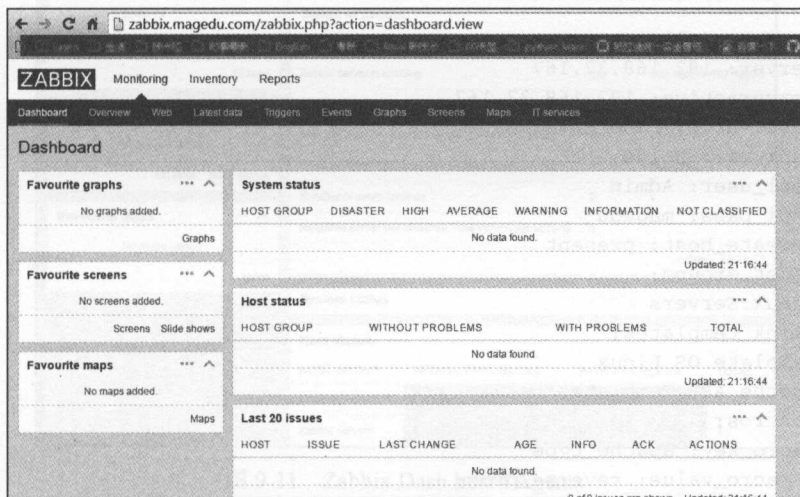


图 9-10 Zabbix Dashboard

因为还没有 Agent 汇报数据，所以图 9-10 中我们看到 Host 和 Group 列表都是空的。接下来部署完 Zabbix Agent 后，我们会看到 Agent 列表。

9.5.2 Zabbix Agent 基于 Ansible 的自动化实现

我们同样使用 Ansible Galaxy 的 dj-wasabi.zabbix-agent 完成 Zabbix Agent 的安裝配置工

作。该 Roles 支持 RedHat、Debian、Ubuntu、opensuse 系统，支持 Zabbix 2.2、2.4、3.0 版本，支持的 DB 类型有 MySQL、PgSQL，不依赖其他第三方。

Agent 的配置相对简单，具体操作步骤如下。

步骤 1：从 Ansible-galaxy 下载并安装 dj-wasabi.zabbix-agent。

```
ansible-galaxy install dj-wasabi.zabbix-agent
```

步骤 2：设置系统 SELINUX 权限为 permissive 或 disabled，不然启动服务会有权限问题。

```
# 临时生效
setenforce 0
# 永久生效
sed -i 's/SELINUX=enforcing/SELINUX=disabled/' /etc/selinux/config
```

步骤 3：编辑 Inventory 文件 (/etc/ansible/hosts)，配置 zabbix-server。添加如下内容：

```
[zabbix-agent]
192.168.37.167
192.168.37.159
```

步骤 4：编辑 zabbix-agent/group_vars/all 变量文件及 zabbix-agent/main.yml 主配置文件。

zabbix-agent/group_vars/all 内容如下：

```
cat group_vars/all
agent_server: 192.168.37.167
agent_serveractive: 192.168.37.167
zabbix_url: http://zabbix.magedu.com
zabbix_api_use: False
zabbix_api_user: Admin
zabbix_api_pass: magedu
zabbix_create_host: present
zabbix_host_groups:
  - Linux Servers
zabbix_link_templates:
  - Template OS Linux
  - Apache APP Template
zabbix_macros:
  - macro_key: apache_type
    macro_value: reverse_proxy
```

该 YML 定义 Zabbix Server 信息，Zabbix 访问地址及用户名密码信息，是否使用 Zabbix API 模板等。

zabbix-agent/main.yml 内容如下：

```
---
- hosts: zabbix-agent
  remote_user: root
  tasks:
```

```
- name: Pip install zabbix-api
  pip: name=zabbix-api

- hosts: zabbix-agent
  remote_user: root
  roles:
    - role: dj-wasabi.zabbix-agent
```

步骤 5：部署安装 Zabbix-agent。

ansible-playbook main.yml

至此，Zabbix Agent 部署完毕，大家可以打开 <http://zabbix.magedu.com/>，并添加对应（如何添加请参考 Zabbix Quickstart[Ⓐ]）的 Agent，并查看其状态是否正常。如一切正常，可以看到 Zabbix Dash board Agents，如图 9-11 所示。

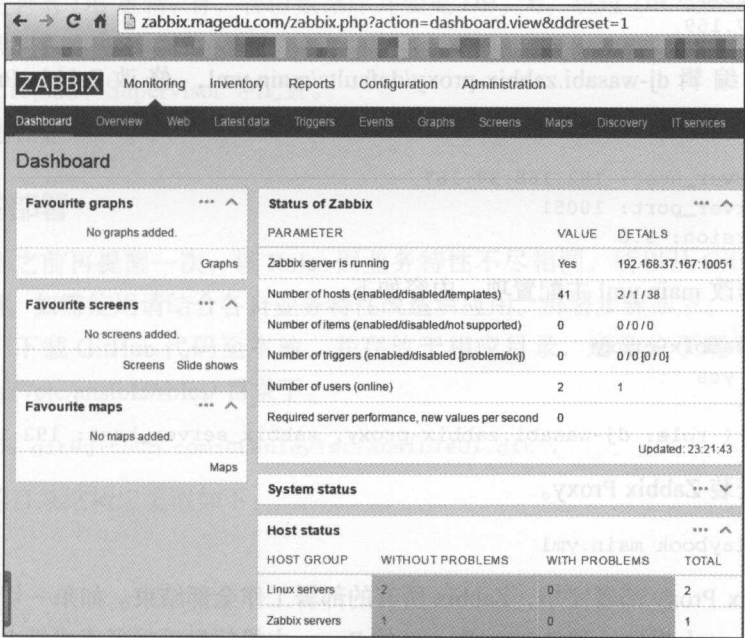


图 9-11 Zabbix Dash board Agents

当然，这里更完善的做法是配置 Zabbix Server 为自动发现，当扫描到指定网段有新主机时即自动添加到 Hosts/Groups 列表。大家可自行研究完善，这些优化点不一一介绍。

9.5.3 Zabbix Proxy 基于 Ansible 的自动化实现

Zabbix Proxy 在 Zabbix 套件中属于可选组件，常用于分布式监控环境中，代理 Server 收

Ⓐ Zabbix Quickstart 网址：<https://www.zabbix.com/documentation/3.0/manual/quickstart/host>。

集部分被监控数据，并统一发往 Server 端。在企业服务器数量较多、Zabbix Server 遇到性能流量 IO 等瓶颈时多会用到。

Zabbix Proxy 的配置我们依然采用 Ansible Galaxy 的 `dj-wasabi.zabbix-proxy` 来完成其部署配置工作。该 Roles 支持 Debian、EL、Ubuntu 系统，不依赖其他第三方。下面我们看具体的安装过程。

步骤 1：从 Ansible-galaxy 下载并安装 `dj-wasabi.zabbix-proxy`。

方式一：

```
ansible-galaxy install dj-wasabi.zabbix-proxy
```

方式二（源库直接使用会报错，所以推荐该方式）：

从 GitHub 下载同名库至本地指定的 Roles 目录下。

步骤 2：编辑 Inventory (/etc/ansible/hosts)，添加 Zabbix Proxy 主机配置。

```
[zabbix-proxy]
192.168.37.159
```

步骤 3：编辑 `dj-wasabi.zabbix-proxy/defaults/main.yml`，修改 Zabbix Proxy 和 Zabbix Server 连接配置。选项较多，如下为必改项：

```
zabbix_server_host: 192.168.37.167
zabbix_server_port: 10051
zabbix_version: 3.0
```

步骤 4：修改 `main.yml` 主配置项。内容如下：

```
- hosts: zabbix-proxy
  sudo: yes
  roles:
    - { role: dj-wasabi.zabbix-proxy, zabbix_server_host: 192.168.37.167 }
```

步骤 5：安装 Zabbix Proxy。

```
ansible-playbook main.yml
```

到此 Zabbix Proxy 部署完毕，Zabbix 所有的部署工作全部结束。如果一切顺利，那么修改完 Zabbix Agent 上报地址和 Zabbix Server 的 Proxy 主机信息后就可正式投入使用。如果不是那么顺利也没有关系，简单的报错可以通过前面所学知识解决。

9.6 Ansible+Git+GitLab 实现自动化发布

自动化发布几乎是所有运维或早或晚都会遇到且需自我消化的一大难题，在自动化运维呼声高涨的时代，每个运维人员对自动化都有自己的理解，也需要有自己的见解。自动化理念决定着运维人员有没有好日子过，自动化程度决定着运维人员日子能过得多好。

本节为大家介绍 Ansible 结合 Git 和 GitLab 实现业务发布自动化的完整案例，该案例可能是本章实战案例中最复杂的，但也是本章介绍的最简单的案例，究其根本自动化发布与各家公司的业务独特性息息相关，每家公司不能一概而论，这里不浪费篇章，只提供实现思路供大家参考。

9.6.1 架构概览

ReleaseAutomation 整体架构如图 9-12 所示。该项目帮助运维人员实现一键自动化发布功能，但又可满足任意步骤都可独立分拆运行，且支持简单的用户交互功能。具体的功能实现如下。

- git pull 从 GitLab 拉取指定分支代码。
- 分发代码至对应主机相应目录。
- 判断是否有 DB 更新：有，备份数据库并变更 DB；无，跳过 DB 变更并跳过 DB 备份。
- 更新代码。
- 生成 Crontab、Supervisor 等配置。
- 重启进程。

9.6.2 架构部署

开始部署之前再提醒一次，该 Roles 因业务特性不尽相同，所以从 GitHub 下载的代码无法直接使用，如需使用请结合各自业务特性改造后应用。部署步骤如下。

步骤 1：下载 GitHub 代码至本地，并存放于相应目录，通常位于现有代码主目录的 roles/ 目录下或 /etc/ansible/roles/ 目录下。

```
git clone git@github.com:stanley1st/ansibleUI.git .
```

下载后的目录结构应类似如下：

```
cp.yml
gw.yml
java-order.yml
java-settle.yml
roles/
├── build
├── composer
├── create_tar
├── crond
├── ...
├── mysql
├── tomcat
└── updatefile
```

步骤 2：编辑 group_vars/all，配置项目 Git 地址及项目文件存放地址。

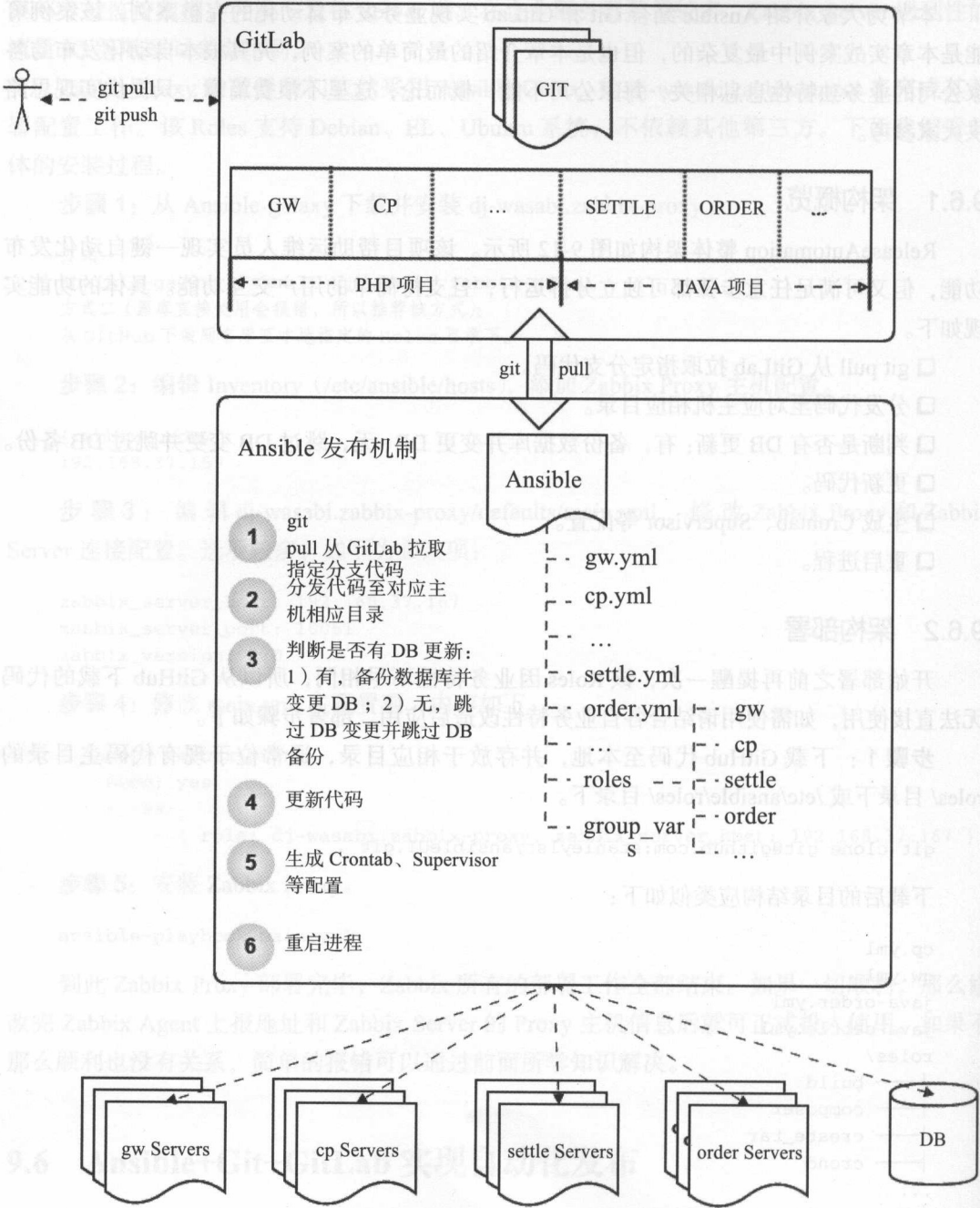


图 9-12 ReleaseAutomation 整体架构

文件中的变量针对所有主机和组有效

```

flag: king
javaflag: java
tech: php
damn: '-'
git: git
package_dir: /srv/deploy/
project_dir: /srv/www/
git_commit: 084295f64f4cb953744a8dd71d9abe1c549a0109
king_gw_path: /srv/www/king-gw/
http_port: 80
repository_gw: git@git.magedu.com:php/king-gw.git
repository_cp: git@git.magedu.com:php/king-cp.git

```

Java 项目配置

```

java_settle_package_dir: /opt/java/
java_settle_project_dir: /opt/webapps
java_package_dir: /opt/java
java_project_dir: /opt/webapps
repository_java: git@git.magedu.com:release-java/release
repository_java_settle: git@git.magedu.com:release-java/release-settle.git
repository_java_order: git@git.magedu.com:release-java/release-order.git

```

步骤 3: 项目发布方式如下。

gw 项目发布方式如下

```

ansible-playbook gw.yml --extra-vars "project=gw git_commit=b7e7ec76efbf2a317c7
1f247c2b8242362b00ae4" --tags="update_pre,process"

```

cp 项目发布方式如下

```

ansible-playbook cp.yml --extra-vars "project=cp git_commit=48c2f2352d65d88062b
89b503781f8cf598cac4b" --tags="update_pre,process"

```

order 项目发布方式如下

```

ansible-playbook java-order.yml --extra-vars "project=order git_commit=cc2033bb
b2d1cc5ff64cd6bda26f5e124bd18f2e" --tags="update_pre,pstop,process,pstart"

```

settle 项目发布方式如下

```

ansible-playbook java-settle.yml --extra-vars "project=settle git_commit=4d9e29
fe52ea5d55296287e0ef6dcaa2872fe26b" --tags="update_pre,pstop,process,pstart"

```

本案例着重为大家介绍自动化发布思路，具体的代码请从 GitHub 下载阅读。需要单独特别说明的是发布方式：

- 笔者公司的 PHP 和 Java 团队分属两个团队，抛开团队管理因素不提，尽管两种语言不同风格，我们通过 Ansible 也实现了发布方式完全相同，对操作者而言学习成本最小化，无论哪种自动化工具均是如此；
- 通过 git_commit 变量可指定发布更新任意版本；
- 通过 --tags 模块化，可指定运行 Tasks 中的任意步骤。

关于自动化发布案例介绍到此结束，但该方式依旧停留在脚本化层面，我们所有的自动化发布工具要多考虑操作者学习成本，因此在后续的第 10 ~ 11 章也会为大家介绍 Ansible 的 Web 化自动发布实现，但需具备一定 Python、Django 及简单前端基础，深一点，能比想象中得到的更多。

9.7 Docker 的 Ansible 自动化应用

Docker 是 PaaS 供应商 dotCloud 开源的一个基于 LXC 的高级容器引擎，源代码托管在 GitHub 上，基于 Go 语言开发，并遵从 Apache 2.0 协议开源。本章不深入探讨 Docker 及 Linux 容器工作原理，但会基于当前两大前沿技术为大家带来两者结合后的最佳技术实践，如构建、管理、部署容器。当然，开始之前我认为各位已经熟读并已安装 Docker^①的相关服务。

9.7.1 Docker 容器入门

我们通过几组简单的案例来了解下 Docker 的工作方式。我们会先使用 Dockerfile 来创建容器，然后 Ansible 通过 docker build 方式创建容器。

编辑文件名为 Dockerfile 的配置文件，内容如下：

```
# 创建 Docker 容器模板
FROM busybox
MAINTAINER Stanley <stanley1st@163.com>

# 容器启动时运行的命令
CMD ["/bin/ping www.magedu.com"]
```

该容器只执行了 ping 命令，但不影响我们后续验证工作，我们只需要对比 Docker 和 Ansible 使用方式间差别即可。在该 Dockerfile 同级目录下，使用如下命令创建自己的 DockerContainer。

```
docker build -t test .
```

执行 docker images 可以看到类似如下返回结果：

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL SIZE
test	latest	7494eee30ea1	3 minutes ago	196.7 MB
<none>	<none>	3cb6b23285d2	5 minutes ago	1.113 MB
<none>	<none>	9705d8b4ac0f	27 minutes ago	1.113 MB
centos	latest	2a332da70fd1	5 days ago	196.7 MB
busybox	latest	437595becdeb	11 weeks ago	1.113 MB

运行创建的容器。

```
docker run --name=test test
```

通过 docker ps -a 命令查看所有容器当前运行状态。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

① Docker 安装网址：<https://docs.docker.com/engine/installation/>。


```
98b25de13a1b test "/bin/ping www.maged 5 minutes ago test
```

通过 CONTAINER ID (98b25de13a1b) 或者 CONTAINER NAME (test) 可以操控以下容器。

□ 启动容器: `docker start [container]`

□ 停止容器: `docker stop [container]`

□ 删除容器: `docker rm [container]`

对于 Docker 的简单介绍到此已经足够了, 我们做个简要总结。

□ Dockerfiles: 通过 Dockerfile 中预定义的配置生成容器。

□ `docker build`: 创建 Dockerfile 生成容器镜像。

□ `docker images`: 列出本地所有容器镜像。

□ `docker run`: 运行并生成新容器镜像。

□ `docker ps -a`: 列出所有运行 / 停止状态的容器状态信息。

□ 关于 Ansible 和 Docker 的结合方式额外需要说明的是, 使用 Dockerfile 生成自定义的容器镜像, 使用 Docker CLI 更为合适, 但最终的镜像部署、应用起停等工作与 Ansible 的结合能更有效地提高工作效率。

9.7.2 使用 Ansible 创建和管理容器

Ansible 内置 docker 模块, 用以管理各生命周期的容器, 接下来的内容将展示其与 Docker 的结合使用方式, 通过 Ansible 完成 9.7.1 节 Docker 的创建、运行、删除。进行下面的内容前请确认 `docker-py` 模块已安装, Ansible Docker 模块依赖其执行 Playbook。如没有安装请运行命令: `pip install docker-py`。

Playbook 的目录结构编排如下:

```
docker/
  main.yml
  test/
    Dockerfile
```

Dockerfile 请参考上节, 其内容不变。编辑 `main.yml` 内容如下 (内容无行号):

```
---
- hosts: localhost
  connection: local

  tasks:
    - name: Build Docker image from Dockerfiles.
      docker_image:
        name: test
        path: test
        state: build
```

该 Playbook 使用 Ansible 内置的 `docker_image` 模块，指定 `test` 目录下的 `Dockerfile` 创建名为 `test` 的容器。运行命令 `ansible-playbook main.yml` 即可完成容器的创建。使用 `docker images` 命令可查看列表中名为 `test` 的容器。

执行 `docker ps -a` 会发现 `test image` 非启动状态，可通过如下 Ansible Playbook 更改其状态：

```
- name: Run the test container.
  docker:
    image: test:latest
    name: test
    state: running
```

执行完毕如上 Playbook 后，再次运行 `docker ps -a`，可以看到 `test` 容器的状态信息。同时，我们也可以通过设置 `state` 状态为 `absent` 来删除 `test` 容器。

9.7.3 基于 Ansible 创建 Flask 的 Docker 容器

我们来创建一些功能更强大的 Docker 容器：一个容器运行我们的 App 应用（内置 Flask，轻量级的 Python Web 构架），一个容器运行数据库（MySQL），还有一个数据存储容器。之所以需要专门的数据存储容器是因为 MySQL 存储的数据需要永久保存，但容器重启后 MySQL 的数据会丢失。本次的内容架构 Docker stack for Flask App 如图 9-13 所示。

本次使用 Ubuntu 14.04 系统作为示例，Docker 的 Playbook 使用 `angswad.docker_ubuntu` 案例，执行 `ansible-galaxy install angswad.docker_ubuntu` 下载 Roles 至本地。在 `docker/provisioning` 目录下编辑 `main.yml` 内容如下：

```
---
- hosts: all
  sudo: yes

  roles:
    - role: angswad.docker_ubuntu

  tasks:
    - include: setup.yml
    - include: docker.yml
```

`sudo: yes` 是 Docker 的安装需求，当前用户属主信息需变更为 `docker` 组。Angswad 的 `docker_ubuntu` Roles 已很完善，无需额外配置。再来看下同级目录下的 `setup.yml` 内容：

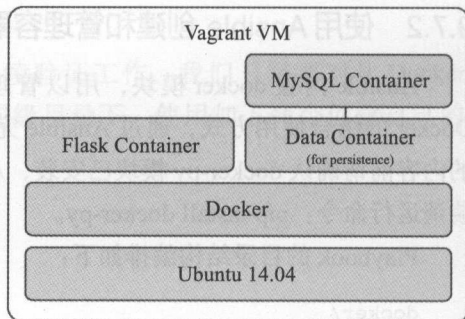


图 9-13 Docker stack for Flask App

```

---
- name: Install Pip.
  apt: name=python-pip state=present
  sudo: yes

- name: Install Docker Python library.
  pip: name=docker-py state=present
  sudo: yes

```

Ansible 通过 Python 的 docker-py 模块来操控 Docker，所以要事先安装该模块。同级目录下的 docker.yml，其核心任务就是创建存储、App 应用、MySQL 容器。

```

---
- name: Build Docker images from Dockerfiles.
  docker_image:
    name: "{{ item.name }}"
    tag: "{{ item.tag }}"
    path: "/vagrant/provisioning/{{ item.directory }}"
    state: build
  with_items:
    - { name: data, tag: "data", directory: data }
    - { name: www, tag: "flask", directory: www }
    - { name: db, tag: "mysql", directory: db }

```

该 YML 调用 docker_image 模块，使用 name、path、state 和 tags 定义每个镜像的属性。Tags 属性类似 git 的 tags 功能，允许为每个镜像标上特殊意义的标识。这里的 YML 创建了 data、www、db 三个镜像，镜像存放于 /vagrant/provisioning/ 下对应的镜像名目录。

这里我们只是创建了 Images，接下来启动所有镜像，即设置 state 状态为 present。编辑 docker_run.yml 内容如下：

```

# 数据容器非 running 状态也可被使用
- name: Run a Data container.
  docker:
    image: data:data
    name: data
    state: present

- name: Run a Flask container.
  docker:
    image: www:flask
    name: www
    state: running
    command: python /opt/www/index.py
    ports: "80:80"

- name: Run a MySQL container.
  docker:
    image: db:mysql

```

```

name: db
state: running
volumes_from: data
command: /opt/start-mysql.sh
ports: "3306:3306"

```

针对 Flask 容器，我们不仅需要保证 App 正常运行，也需要容器正常运行。所以在该容器启动时我们设置运行如下命令：

```
command: python /opt/www/index.py
```

此外，我们通过 ports: "80:80" 对外映射 80 端口，以供外部用户通过 HTTP 的方式访问 Flask 应用，该功能相当于 Docker 的 --publish 参数。

Flash 容器只是对外提供 Web 访问，没有永久性数据存储需求，所以容器的启停对其影响不大，但对 MySQL 需永久性保留用户数据的需要是致命的，容器的起停会导致数据丢失，所以对 MySQL 容器我们额外配置 data 容器，额外挂载专门的数据硬盘。针对 db 容器，我们额外指定 volumes_from 和 command 参数，volumes_from 挂载指定容器的数据卷，command 指定容器启动后运行 Shell 脚本来启动 MySQL。

到目前为止，我们的目录结构应该如下：

```

docker/
├── main.yml
├── provisioning
│   ├── data
│   ├── db
│   ├── docker_run.yml
│   ├── docker.yml
│   ├── main.yml
│   ├── setup.yml
│   └── www

```

每个容器都有自己对应的项目目录。

9.7.4 数据存储容器配置

数据存储容器的配置不需要做太多事情，只需创建一个数据存储目录并对外可 mount，以便于其他容器使用 VOLUME 正常读取数据。创建 docker/provisioning/data/Dockfile 文件内容如下：

```

# 用 Docker 容器创建数据卷
FROM busybox
MAINTAINER Jeff Geerling <geerlingguy@mac.com>

# 为 MySQL 创建数据卷
RUN mkdir -p /var/lib/mysql

```



```
VOLUME /var/lib/mysql
```

该容器启动时会 RUN `mkdir -p /var/lib/mysql`，创建 `/var/lib/mysql` 的目录，并指定其 VOLUME 属性。

9.7.5 Flask 容器配置

Flask 是基于 Python 的轻量级 Web 框架，类似于 PHP，其他后端需连接 DB 服务。这里我们也编辑一个简单的 Web 页面，用以展现连接状态。编辑 `docker/provisioning/www/index.py.j2` 内容如下：

```
# 构建测试页面
from flask import Flask
from flask import Markup
from flask import render_template
from flask.ext.sqlalchemy import SQLAlchemy

app = Flask(__name__)

# 配置 MySQL 连接
db = SQLAlchemy()
db_uri = 'mysql://admin:admin@{{ host_ip_address }}/information_schema'
app.config['SQLALCHEMY_DATABASE_URI'] = db_uri
db.init_app(app)

@app.route("/")
def test():
    mysql_result = False

    try:
        if db.session.query("1").from_statement("SELECT 1").all():
            mysql_result = True
    except:
        pass

    if mysql_result:
        result = Markup('<span style="color: green;">PASS</span>')
    else:
        result = Markup('<span style="color: red;">FAIL</span>')

    # 返回页面和结果
    return render_template('index.html', result=result)

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=80)
```

该 Python 脚本定义监听本机所有 IP 的 80 端口，并生成 MySQL 连接状态的简明 HTML 页面。在开始运行 Playbook 之前，我们还需定义 `docker/provisioning/www/templates/index.html` 模

板文件。

```
<!DOCTYPE html>
<html>
<head>
  <title>Flask + MySQL Docker Example</title>
  <style>* { font-family: Helvetica, Arial, sans-serif }</style>
</head>
<body>
  <h1>Flask + MySQL Docker Example</h1>
  <p>MySQL Connection: {{ result }}</p>
</body>
</html>
```

其中的 `{{ result }}` 变量信息会被 Jinja 替换为 MySQL 的连接状态信息。

App 应用配置完毕后，我们接着配置容器来运行该 App。接着定义的 Dockerfile 将安装一些必需的依赖包、复杂 Ansible 的配置文件的指定目录，并运行 Ansible-playbook 命令等操作。

```
# Flask 容器
FROM ansible/ubuntu14.04-ansible
MAINTAINER Jeff Geerling <geerlingguy@mac.com>

# 安装 Flask 应用的依赖程序
RUN apt-get install -y libmysqlclient-dev python-dev
RUN pip install flask flask-sqlalchemy mysql-python

# 安装并运行 Playbook
COPY playbook.yml /etc/ansible/playbook.yml
COPY index.py.j2 /etc/ansible/index.py.j2
COPY templates /etc/ansible/templates
RUN mkdir -m 755 /opt/www
RUN ansible-playbook /etc/ansible/playbook.yml --connection=local

EXPOSE 80
```

这里没有使用 Ansible 内置的 apt 和 pip 安装软件包，相反我们使用 RUN 命令，这有利于 Docker 缓存这些命令。一般而言，软件包的安装配置越复杂，Ansible 的内置功能会使其安装配置越简单，但这里我们使用 RUN 命令是为了在 docker build 时减少漫长的等待时间。

Dockerfile 的末尾，我们运行 Playbook 对外开放本机 80 端口以供外部访问。接着我们配置 App 的部署 Playbook。Flask 容器需获取 host_ip_address，用以替换 index.py.j2 中的变量，该功能在 docker/provisioning/www/playbook.yml 实现。

```
---
- hosts: localhost
  sudo: yes
```

```

tasks:
- name: Get host IP address.
  Shell: "/sbin/ip route|awk '/default/ { print $3 }'"
  register: host_ip
  changed_when: false

- name: Set host_ip_address variable.
  set_fact:
    host_ip_address: "{{ host_ip.stdout }}"

- name: Copy Flask app into place.
  template:
    src: /etc/ansible/index.py.j2
    dest: /opt/www/index.py
    mode: 0755

- name: Copy Flask templates into place.
  copy:
    src: /etc/ansible/templates
    dest: /opt/www
    mode: 0755

```

首先通过 Shell 模块获取 IP 信息，并赋予 register 的自定义变量 host_ip。然后通过 set_fact 设置 host_ip_address 变量全局可用。最后两条 Tasks 复制模板文件至指定位置。现在 docker/provisioning/www 的目录架构应该类似如下：

```

www/
  templates/
    index.html
  Dockerfile
  index.py.j2
  playbook.yml

```

再配置 MySQL 容器就可以运行并查看结果了。

9.7.6 MySQL 容器配置

本节介绍的重点是：如何配置 MySQL 在 Docker 容器中运行及如何挂载 data 容器的 VOLUME。

首先使用 geerlingguy.mysql Roles，配置 MySQL（安装 geerlingguy.mysql : ansible-galaxy install geerlingguy.mysql）。

```

---
- hosts: localhost
  sudo: yes

vars:

```

```
mysql_users:
  - name: admin
    host: "%"
    password: admin
    priv: "*.*:ALL"

roles:
  - geerlingguy.mysql
```

该 YML 使用 `mysql_users` 变量设置 MySQL User 信息，使用 `geerlingguy.mysql` 配置 MySQL，编辑完毕后请保存至 `docker/provisioning/db/playbook.yml`。大家是否还记得，因为容器重启会导致数据丢失，所以我们专门构建了一个 `db` 容器用以挂载 MySQL 的数据盘，但如果在某些情况下启动时该数据盘不存在该如何处理呢？这也是我们前面创建 MySQL 容器时没有直接以 `Daemon` 方式启动 MySQL 而是使用 `/opt/start-mysql.sh` 启动脚本启动的原因。

`docker/provisioning/db/start-mysql.sh` 脚本内容如下：

```
# !/bin/bash

# 如果连接新的数据库，需要重新设置 MySQL
if [[ ! -d /var/lib/mysql/mysql ]]; then
  rm -f ~/.my.cnf
  mysql_install_db
  ansible-playbook /etc/ansible/playbook.yml --connection=local
  mysqladmin shutdown
fi
```

该脚本会检查是否存在 `/var/lib/mysql/mysql` 目录，如果不存在会删除 `my.cnf` 配置文件，初始化 MySQL 配置并运行 `playbook.yml` 关闭 MySQL。最后再 `mysqld_safe` 重新启动 MySQL。在 `Dockerfile` 中我们需要确保 `playbook.yml` 和 `playbook.yml` 的存放在正确的目录中。编辑 `docker/provisioning/db/Dockerfile` 内容如下：

```
# MySQL 容器
FROM ansible/ubuntu14.04-ansible
MAINTAINER Jeff Geerling <geerlingguy@mac.com>

# 通过 Galaxy 安装 MySQL
RUN ansible-galaxy install geerlingguy.mysql

# 复制启动脚本至 /opt/
COPY start-mysql.sh /opt/start-mysql.sh
RUN chmod +x /opt/start-mysql.sh

# 安装并运行 Playbook
COPY playbook.yml /etc/ansible/playbook.yml
RUN ansible-playbook /etc/ansible/playbook.yml --connection=local
```


EXPOSE 3306

该 Dockerfile 指定基于 Ubuntu 14.04 创建容器镜像，同时通过 Ansible-galaxy 下载 MySQL 的 Roles，所以整个过程需保证服务器可访问互联网。配置最后指定开放 3306 端口以供程序访问。至此，整体 docker/provisioning/db 的目录结构应该如下：

```
docker/provisioning/db/
├── Dockerfile
├── playbook.yml
└── start-mysql.sh
```

至此，繁琐的配置结束，终于到检查成果的时候了！

9.7.7 启动容器

切换至 docker 项目主目录，确保所有的容器是正常启动状态，绑定本机 HOST 配置域名解析，如：

```
192.168.37.159 docker.dev
```

打开浏览器你会访问如图 9-14 所示的 Docker Ansible 页面。

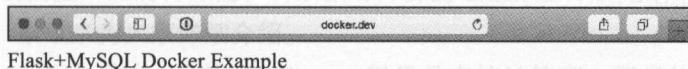


图 9-14 Docker Ansible

如果你能顺利进行到这步并正常访问如图 9-14 所示的 Docker Ansible 的页面，那么说明你和我的环境相近，已经坚持到这里，相信你至少已经掌握 Ansible 应该以何种方式与 Docker 相结合。整体过程中遇到最多的问题可能有如下几个：

- ☐ Npm 安装软件包超时或异常。
- ☐ GitHub 无法访问。
- ☐ Ansible-galaxy install 无法下载 Roles 至本地。
- ☐ Ansible Docker-py 调用的 Client 版本高于 Server 版本。
- ☐ 基于要重新下载 Ubuntu 系统。

这些问题都可以搜索 Google 网站获取答案，学会解决问题也是一项必备技能。

9.8 本章小结

本章着重为大家介绍笔者日常工作中 Ansible 的运用方式、应用场景及心得体会。考虑部分新手零基础或缺 Python 开发经验，本章从最基础的 SSH 批量认证介绍起，该部分既包括

新手最基础的 `ssh-copy-id` 方式，又涵盖 Python Paramiko 多线程的高级开发实现，并且还有企业通用高可用架构案例、当今日益流行的 ELK 分布式日志系统、轻量级 Inotify+Rsyncd+log.io 日志时时跟踪系统、业内著名的 Zabbix 分布式监控系统、Ansible 结合 Git+GitLab 实现业务自动化发布、声名大噪的 Docker 容器技术应用。这些案例从底层新手所需的 Linux 系统认证，至应用层资深工程师所需的高可用架构、高级日志系统、自动化发布系统、容器技术均有涉及，几乎囊括了运维日常所需技能的各个方面，相信大家看完本章会收获颇丰。

Ansible 基于 Windows 的管理架构

如第 1 章介绍，作为关注度最高的集中化管理工具，Ansible 同样支持 Windows 系统，因为 Windows 系统商业产品的特殊性，所以相对开源的 Linux 发行版无论是配置还是管理方式都有较大差别，本章来为大家详细介绍。

Ansible 从 1.7+ 版本开始，支持 Windows，但只是支持被管理，即只能作为远程主机端被 Ansible 远程调度使用，所以管理机必须为 Linux 系统。基于 Windows 系统的通信方式不同于 Linux 系统，远程主机的通信方式由 SSH 变更为 PowerShell。Windows 是图形化操作系统，远程连接依赖自带的远程连接工具，自身并不支持 SSH，但在 Linux 服务器领域迅猛发展的冲击下，Windows 也折腾出来自己的 SSH 工具：PowerShell。最新的 PowerShell 不仅支持 SSH，甚至还将 PowerShell 开源，这真是 Windows 运维人员福音呀！

微软自 Windows 2000 操作系统开始，其域功能的账户认证方式基于 Kerberos，且该方式一直延续至今，因此管理机必须预安装 Python 的 Winrm^① 模块，方可和远程 Windows 主机正常通信。但 PowerShell 需 3.0+ 版本及 Management Framework 3.0+ 版本，实测 Windows 7 SP1 和 Windows Server 2008 R2 及以上版本系统经简单配置可正常与 Ansible 通信。简单总结如下：

- 1) 管理机必须为 Linux 系统且需预安装 Python Winrm 模块；
- 2) 底层通信认证协议基于 Kerberos^②，Windows 使用的连接工具为 PowerShell 而非 SSH；

① winrm 全称是 Windows Remote Management，是 Windows 基于 Python 的管理模块。支持诸如 BasicAuthentication、DigestAuthentication、NegotiateAuthentication、KerberosAuthentication 等多种标准身份认证和信息加密。

② Kerberos：网络认证协议的一种。

3) 远程主机 PowerShell 版本为 3.0+, Management Framework 版本为 3.0+。
以上条件满足后,方可正常与 Ansible 通信。下面我们逐步深入介绍其部署安装及使用。

10.1 Ansible 管理机部署安装

如本章伊始介绍,Windows 系统无法作为管理机,只能作为远程主机被管理,且管理机需预先安装 Python 的 Winrm 模块,使用的安装命令如下:

```
pip install "pywinrm>=0.1.1"
```

如远程的 Windows 主机没有入域^①(Windows 活动目录域服务,详见官网),可跳过下方的步骤 1 和步骤 3 操作。如远程 Windows 主机是基于 Active Directory (简称 AD) 的管理方式,管理机和远程主机基于 Kerbero 认证,所以管理机需额外安装 python-kerbero 和 MIT krb5 依赖库。

步骤 1: 安装 python-kerberos 依赖,命令如下(不同发行版本请参考不同的命令格式)。

□ YUM 方式(Centos、RedHat、Fedora)

```
yum -y install python-devel krb5-devel krb5-libs krb5-workstation
```

□ Apt 方式(Ubuntu)

```
sudo apt-get install python-dev libkrb5-dev
```

□ Portage 方式(Gentoo)

```
emerge -av app-crypt/mit-krb5
emerge -av dev-python/setuptools
```

□ pkg 方式(FreeBSD)

```
sudo pkg install security/krb5
```

□ OpenCSW 方式(Solaris)

```
pkgadd -d http://get.opencsw.org/now
/opt/csw/bin/pkgutil -U
/opt/csw/bin/pkgutil -y -i libkrb5_3
```

□ Pacman 方式(Arch Linux)

```
pacman -S krb5
```

步骤 2: 安装 python-kerberos, OSX 和 Linux 发行版均已默认安装。如需安装请使用如

① Windows AD 域网址: [https://msdn.microsoft.com/en-us/library/aa362244\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/aa362244(v=vs.85).aspx)。

下命令。

```
pip install Kerberos
```

步骤 3: 配置 Kerberos, 参考如下内容修改配置 `/etc/krb5.conf`。

```
[realms]
  MY.DOMAIN.COM = {
    kdc = domain-controller1.my.domain.com
    kdc = domain-controller2.my.domain.com
  }
```

之后于 `[domain_realm]` 后添加如下内容:

```
[domain_realm]
  .my.domain.com = MY.DOMAIN.COM
```

通过以下命令验证域账户认证情况:

```
kinit user@MY.DOMAIN.COM
```

步骤 4: 同理配置 Inventory 主机信息和 `group_vars/windows.yml` 变量信息。

Inventory (默认 `/etc/ansible/hosts`) 添加如下信息:

```
# 添加 Windows 组, 添加两台主机, 分别为 win1.magedu.com 和 win2.magedu.com
[windows]
win1.magedu.com
win2.magedu.com
```

`group_vars/windows.yml` 添加如下信息:

```
# 指定远程连接的认证用户名
ansible_user: Administrator
# 指定如上 Administrator 用户的连接密码
ansible_password: magedu@beijing
# 指定连接端口, 默认 5986 端口
ansible_port: 5986
# 指定连接方式为 winrm
ansible_connection: winrm
# 忽略交互确认
ansible_winrm_server_cert_validation: ignore
```

至此, 服务端配置完毕, 如需和远程 Windows 正常通信, 仍需对 Windows 做一定配置修改, 这点和 Linux 系统是有区别的。Windows 远程主机的详细配置方式见 10.2 节。

10.2 Windows 系统预配置

和 Linux 发行版稍有区别, 远程主机系统如为 Windows, 需预先如下配置, 方可和管理机通信。

- ❑ 安装 Framework 3.0+;
- ❑ 设置 PowerShell 本地脚本运行权限为 remotesigned;
- ❑ 升级 PowerShell 至 3.0+;
- ❑ 自动设置 Windows 远端管理, 英文全称为 WS-Management。

我们逐一介绍。

1) 安装 Framework 3.0+。

下载链接为:

http://download.microsoft.com/download/B/A/4/BA4A7E71-2906-4B2D-A0E1-80CF16844F5F/dotNetFx45_Full_x86_x64.exe

下载至本地后双击安装即可。期间可能会多次重启, 因电脑配置差异有可能需从微软官网下载依赖组件, 所以安装期间请保证电脑正常连接 Internet。

2) 设置 PowerShell 本地脚本运行权限为 remotesigned。

Windows 系统默认只有 Administrator 用户可执行 SP 脚本, 即使是管理员, 也需额外修改注册表开放 SP 脚本的执行权限。

步骤 1: 打开 CMD, 输入 regedit.exe, 如图 10-1 所示, 打开注册表。

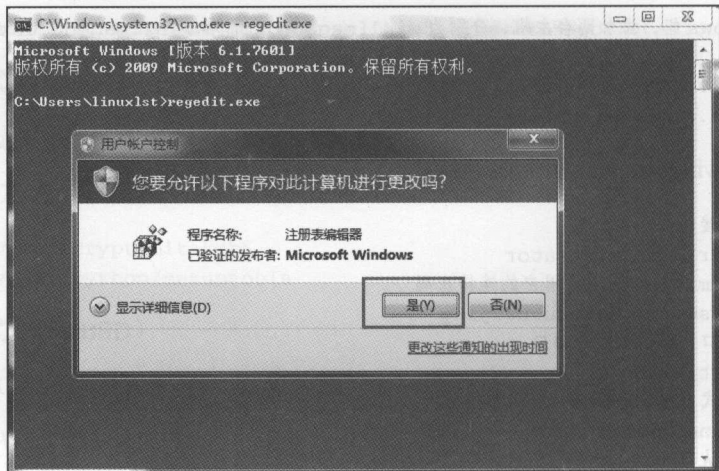


图 10-1 打开注册表

步骤 2: 设置脚本 SP 权限为系统可运行。

依次打开注册表目录 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell, 在如图 10-2 所示的界面中修改 SP 脚本可执行权限为 remotesigned。

或者, 输入 PowerShell 执行命令: set-executionpolicy -executionpolicy unrestricted, 返回结果如图 10-3 所示。

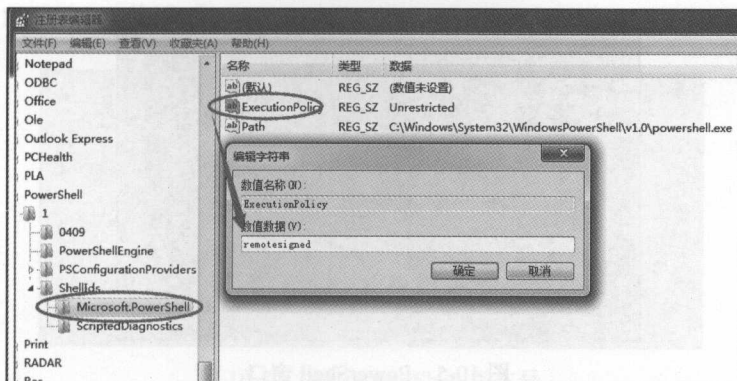


图 10-2 设置 SP 脚本系统可运行

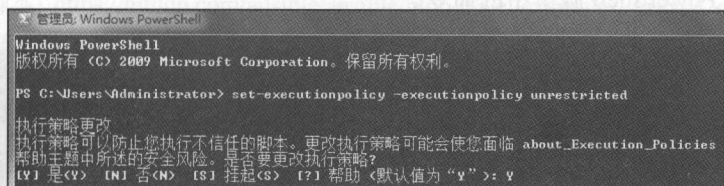


图 10-3 设置 SP 脚本执行策略

这里共有 4 种权限。

- ☐ Restricted, 默认的设置, 不允许任何 script 运行。
- ☐ AllSigned, 只能运行经过数字证书签名的 script。
- ☐ RemoteSigned, 运行本地的 script, 不需要数字签名, 但是运行从网络上下载的 script 就必须要有数字签名。
- ☐ Unrestricted, 允许所有的 script 运行。

我们需要修改权限为第 3 种权限, 即 RemoteSigned。

3) 升级 PowerShell 至 3.0+。

查看 Power 的版本号, 打开 PowerShell 的步骤如图 10-4 所示。单击 Windows 系统键 (即数字 1 的圆形), 在搜索程序和文件 (即数字 2 的方框) 框中输入 powershell, 选择 Windows PowerShell (即数字 3), 之后, 单击即可。

单击后会出现如图 10-5 所示的 PowerShell 窗口。

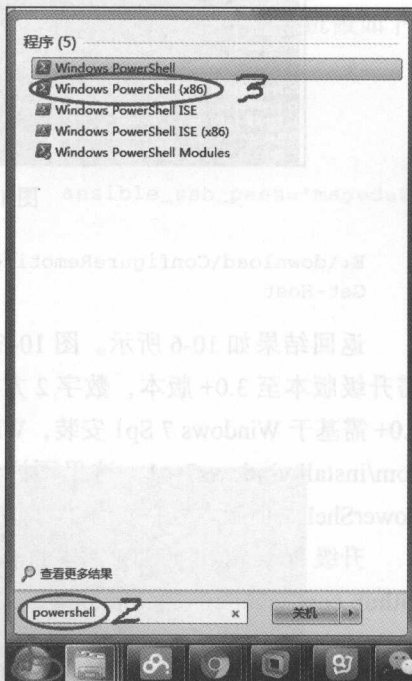


图 10-4 打开 PowerShell 的步骤

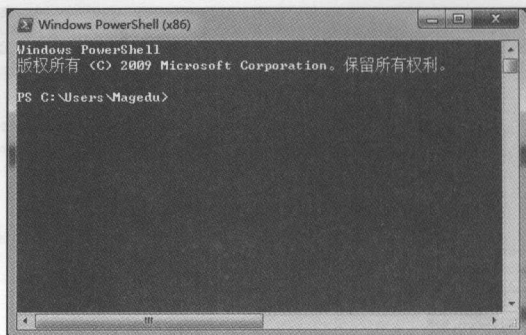


图 10-5 PowerShell 窗口

图中 PS C:\Users\Magedu> 的后面可通过键盘输入所需内容或命令。我们分别输入如图 10-6 所示的查看 PowerShell 版本中的命令。

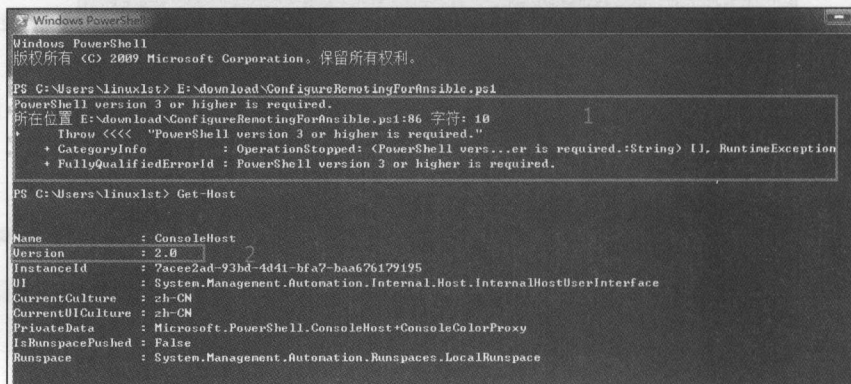


图 10-6 查看 PowerShell 版本

```
E:\download\ConfigureRemotingForAnsible.ps1
Get-Host
```

返回结果如 10-6 所示。图 10-6 中数字 1 方框中的报错部分表示 PowerShell 版本过低，需升级版本至 3.0+ 版本，数字 2 方框中的内容表示当前 PowerShell 版本为 2.0。PowerShell 3.0+ 需基于 Windows 7 Sp1 安装，Windows 7 系统 Sp1 补丁升级请参考 <http://windows.microsoft.com/installwindows7sp1>，这里不详细介绍。Windows 7 和 Windows Server 2008 R2 默认安装 PowerShell，但版本号一般为 2.0 版本，所以这里我们需升级 PowerShell 至 3.0+ 版本。

升级 PowerShell 至 3.0+ 版本的下载网址如下：https://github.com/cchurch/ansible/blob/devel/examples/scripts/upgrade_to_ps3.ps1。下载至本地后，如图 10-7 所示。右键选择“使用 PowerShell 运行”，执行完毕重启系统后，在 PowerShell 窗口执行 Get-Host 命令，结果如

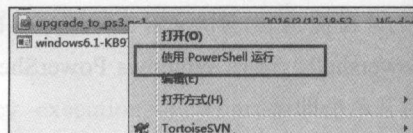


图 10-7 使用 PowerShell 执行 SP 脚本

图 10-8 所示。PowerShell 版本为 3.0，正常。

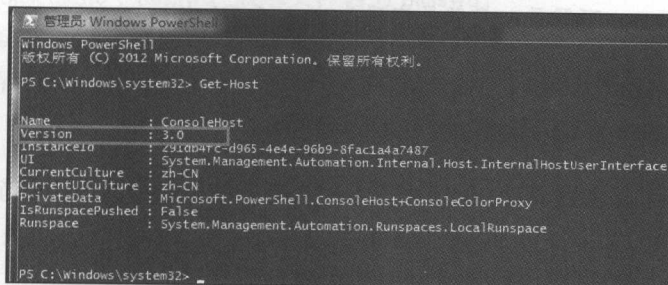


图 10-8 获取 PowerShell 版本号

4) 自动设置 Windows 远端管理 (WS-Management, WinRM)。

下载补丁脚本: <https://github.com/ansible/ansible/blob/devel/examples/scripts/ConfigureRemotingForAnsible.ps1> 至本地，右击后选择“使用 PowerShell 运行”，执行结果没有返回错误即为正常。

如执行后出现“由于此计算机上的网络连接类型之一设置为公用，因此 WinRM 防火墙例外将不运行”类似报错信息，请在 PowerShell (PowerShell 的打开方式) 中执行以下命令，设置为强制执行尝试解决。

```
Enable-PSRemoting -SkipNetworkProfileCheck -Force
```

远程 Windows 主机配置到此结束。我们验证一下配置是否有问题，在 Master 机做如下设置。

步骤 1: 配置 Inventory 添加 /etc/ansible/hosts 配置。

```
[windows]
192.168.37.146 ansible_ssh_user="Administrator" ansible_ssh_pass="magedu@beijing" ansible_ssh_port=5986 ansible_connection="winrm"
```

步骤 2: 在 Master 上执行命令。

```
ansible windows -m win_ping
```

返回结果如图 10-9 所示。

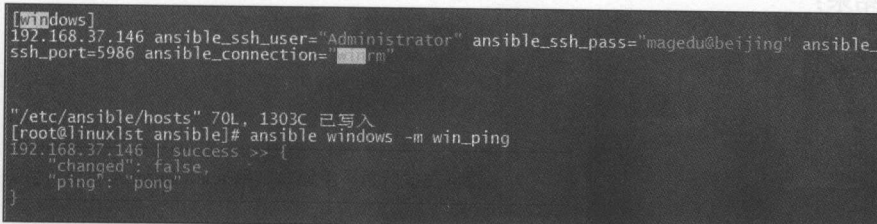


图 10-9 Ansible 连接 Windows 远程主机



Windows 系统建议使用 Administrator 用户，避免不可预知的问题。命令运行前请设置 Administrator 可登录，且登录密码为 magedu@beijing。

10.3 Windows 下可用模块

Windows 下可用模块虽不及 Linux 丰富，但基础功能均包括在内。下面介绍日常工作常用的模块，请参考。

- ❑ win_acl (E): 设置文件 / 目录属主、属组权限；
- ❑ win_copy: 复制文件到远程 Windows 主机；
- ❑ win_file: 创建、删除文件或目录；
- ❑ win_lineinfile: 匹配替换文件内容；
- ❑ win_package (E): 安装 / 卸载本地或网络软件包；
- ❑ win_ping: Windows 系统下的 ping 模块，常用来测试主机是否存活；
- ❑ win_service: 管理 Windows Services 服务；
- ❑ win_user: 管理 Windows 本地用户。

更多模块及详细介绍请参考官网：http://docs.ansible.com/ansible/list_of_windows_modules.html

除 win 开头的模块外，scripts、raw、slurp、setup 模块在 Windows 下也可正常使用。

10.4 节我们会演示 Windows 下的案例。

10.4 Windows Ansible 模块使用实战

本节通过几个实战案例为大家演示一些常用模块的用法。

案例 1: 传输 /etc/passwd 文件至远程 E:\file\ 目录下。

执行命令：

```
# 复制本机 /etc/passwd 文件至远程主机 E 盘的 file 目录下
ansible windows -m win_copy -a 'src=/etc/passwd dest=E:\file\passwd'
```

返回结果：

```
192.168.37.146 | success >> {
  "changed": true,
  "checksum": "896d4c79f49b42ff24f93abc25c38bc1aa20afa0",
  "operation": "file_copy",
  "original_basename": "passwd",
  "size": 2563
}
```

部分返回结果诠释：

□ "operation": "file_copy" 表示执行的操作为 file_copy;

□ "original_basename": "passwd" 表示文件名为 passwd;

□ "size": 2563 表示文件大小为 2563 字节。

Playbook 内容如下:

```
---
- name: windows module example
  hosts: windows
  tasks:
    - name: Move file on remote Windows Server from one location to another
      win_file: src=/etc/passwd dest=E:\file\passwd
```

整体 Playbook 的写法与远程主机为 Linux 时的写法格式没有区别, 只是使用模块有差别, Windows 系统对应的文件传输模块名为 win_file, Linux 系统的文件传输模块名为 file。

案例 2: 删除案例 1 中的文件 E:\file\passwd。

执行命令:

```
ansible windows -m win_file -a "path=E:\file\passwd state=absent"
```

返回结果:

```
192.168.37.146 | success >> {
  "changed": true
}
```

true 返回表示命令执行无误, 登录远程查看可确认 passwd 文件已被删除。

案例 3: 为远程 Windows 主机新增用户 stanley, 密码为 magedu@123, 属组为 Administrators。

执行命令:

```
ansible windows -m win_user -a "name=stanley passwd=magedu@123
group=Administrators"
```

返回结果:

```
192.168.37.146 | success >> {
  "account_disabled": false,
  "account_locked": false,
  "changed": true,
  "description": "",
  "fullname": "stanley",
  "groups": [
    {
      "name": "Administrators",
      "path": "WinNT://WORKGROUP/LINUXLST/Administrators"
    }
  ],
  "name": "stanley",
  "password_expired": true,
  "password_never_expires": false,
```

```

    "path": "WinNT://WORKGROUP/LINUXLST/stanley",
    "sid": "S-1-5-21-3965499365-1200628009-3594530176-1004",
    "state": "present",
    "user_cannot_change_password": false
}

```

部分返回结果诠释。

- account_disabled: 禁用用户登录;
- account_locked: 解锁用户;
- groups: 用户所属组;
- name: 用户名;
- password_expired: 下次登录修改密码;
- user_cannot_change_password: 用户是否可修改密码。

添加的用户可登录远程主机, 右键单击后, 依次选择计算机→管理, 在本地用户和组条目下查看新增的用户信息。关于 Windows 的操作非本书重点, 这里不赘述。

案例 4: 重启 Windows spooler 服务。

Windows 下重启进程使用 win_service 模块来执行远程命令。该模块使用方式如下:

```
ansible windows -m win_service -a "name=spooler state=restarted"
```

返回结果:

```

192.168.37.146 | success >> {
    "changed": true,
    "display_name": "Print Spooler",
    "name": "spooler",
    "start_mode": "auto",
    "state": "running"
}

```

spooler 服务的状态可远程登录后在服务列表中查看其状态 (具体操作这里不赘述)。

10.5 本章小结

本章为大家介绍了当远程主机为 Windows 时 Ansible 的管理机制, 只是 Windows 主机的配置工作增加了很多工作量。很多运维朋友之所使用 Windows, 多数也是因为历史原因和业务缘由。相信在整个配置过程中大家也能感觉到 Windows 配置的复杂程度上相对于 Linux 要麻烦很多, 最关键问题是与 Ansible Agentless 观念相违背。但为了迎合少数 Windows 服务用户, 也只能兼容。在如今服务器市场 Linux 大为流行的背景下, 各软件对 Windows 的支持力度确实不如预期, 也曾收到业界 Windows 应用者反馈 Ansible 管理 Windows 期间存在诸如: 大文件远程传输不完整、Windows 主机较多时部分主机命令执行过程中无响应等意想不到的问题。其实何止 Ansible, 现流行的集中化管理工具对 Windows 的支持力度均一般 (Cygwin 可以尝试)。

Ansible 安全优化篇

对于一台全新安装的服务器，尤其是那些直接面向公网的服务器来说，最重要的一项配置应该就是安全配置了。

针对非授权连接和截取通信信息等攻击行为，我们总结了如下 9 条方法，来避免上述攻击手段可以带来的危害：

- 使用安全加密的通信方式；
- 禁止 root 用户远程登录并充分利用 sudo；
- 移除非必需的软件，只开放需要用到的端口；
- 遵守权限最小化原则；
- 及时更新操作系统和软件；
- 使用合理配置过的、有针对性的防火墙；
- 确保日志文件被及时迁移、存放和切割；
- 监测系统登录情况，封掉可疑的 IP 地址；
- 正确使用 SELinux 和 AppArmor。

大部分情况下，我们的服务器的安全要比我们想象的要低很多。在很多备受瞩目的安全攻击事件中，网络内一台安全性较差的主机就充当了黑客或病毒攻击网络内其他主机的网关。一个好的主机安全策略，可以帮助你获得一名运维工程师至高无上的荣誉——主机零宕机。

本章将会学习到如何借助 Ansible 来加固我们的主机安全性。

11.1 SSH 与远程连接简介

在电脑发明之初，一台计算机的体积和一间房子差不多大。还有指令穿孔卡片不停地被

传进计算机中，另一端的打印机不停地将运算结果印刻在另外一部分卡片上。成千上万个机械部件相互配合着流畅地（不出故障的时候）完成一系列计算任务。

随着科技的进步，计算机的体积变得越来越小，人机交互式操作的终端界面也变得越来越友好，但依然需要有线缆来连接才能通信。直到 20 世纪 60 年代，大型机问世。大型机最初可以通过打字机和电传打字机接口进行输入，后来出现了键盘和小型的文字显示界面。到了 20 世纪七八十年代，网络技术的突飞猛进，远程终端连接管理开始被一些大型的计算机中心运用。

11.1.1 Telnet

Telnet 协议诞生于 20 世纪 60 年代后期，最初被应用到基于 TCP 协议的大型私有网络之中，默认端口并非为 23 号端口，后来逐渐在公网上流行起来。

Telnet 是一种文本协议，用于在不同网络间传输数据。Telnet 属于底层协议，至今它依然是我们现在使用的很多通信协议的基础，比如 HTTP、FTP 以及 POP3。但是，纯文件的数据流在网络中并不安全，即使使用了 TLS 和 SASL 协议进行加密也不安全。随着 SSH 的到来，Telnet 开始逐渐退出远程管理的舞台。

Telnet 现在主要用于配置 serial 端口或用于检测远程主机上的某些服务是否运行正常，比如通过连接 80 端口来测试 HTTP 服务是否正常，连接 3306 端口来检测 MySQL 服务是否正常等。现在主流的 Linux 发行版中，已经不再默认安装 Telnet 软件了。

11.1.2 RLOGIN、RSH 和 RCP

RLOGIN 在 1983 年随着 BSD 的 4.2 版本一同问世，同时被多数的类 UNIX 系统兼容。RLOGIN 在 20 世纪 80 年代到 90 年代期间甚为流行。RLOGIN 和 Telnet 类似，都是通过密码认证来登录远程系统，但是 RLOGIN 同时还支持对信任主机的免密码登录。在功能方面，RLOGIN 也要比 Telnet 更强大一些，RLOGIN 支持很多特定字符和命令，但是 Telnet 在运行和识别这些命令时则需要额外进行转译。

RLOGIN 默认端口是 TCP 的 513 号端口，也使用纯文本通信，所以其安全性也存在問題。1998 年，RLOGIN 的更多内在缺陷被卡内基 - 梅隆大学爆出。

RSH (Remote Shell) 是一个命令行工具，用于单独执行远程 Shell 命令。RCP (Remote Copy) 是用来复制远程文件的。RSH 和 RCP 拥有和 RLOGIN 一样的安全问题，因为它们使用同样的通信方式，只是端口不一样而已。

11.1.3 SSH

SSH (Secure Shell) 诞生于 1995 年，由芬兰人 Tatu Ylönen 开发。当时 Tatu 开发 SSH 的初衷是应对他们学校里的一次密码嗅探攻击 (password-sniffing attack)。他看到了文本通信的安全缺陷，这促使他开发出一款强加密的远程管理工具，也就是现在我们常用的 SSH。

那么 SSH 是如何工作的呢？它比起 Telnet 和 RLOGIN 又有哪些优势呢？这一切都要从最底层的连接方式说起。SSH 的连接加密方式非常类似于 HTTP 的 SSL 加密，同时 SSH 的认证层还增加了更多的安全机制。

1) 当使用命令 `ssh admin@example.com` 连接主机 `example.com` 的时候，当前主机和远程主机 `example.com` 之间是需要进行密钥认证的。

2) 如果是第一次连接一台远程主机，或者从上次连接过之后，主机密钥发生了变化，那么 SSH 会提示你是否批准使用当前主机的密钥进行连接（这通常发生在使用域名进行连接的情况下，使用 IP 地址进行连接不会出现这样的提示）。

3) 如果你在 `~/.ssh` 目录下的私钥和要连接的远程主机上的 `~/.ssh/authorized_keys` 文件中的公钥匹配，那么连接过程将会直接进行下一步。否则，如果远程主机上 SSH 设置了允许密码认证，SSH 将会提示你输入密码。

4) 被传送到远程主机上的公钥会结合其他一些加密方式，如：AES、3DES、Blowfish 等，一起为所有的通信进行加密。

5) 所有连接将会一直维护加密状态，直到用户退出远程连接或者 SSH 所执行的操作完成。比如通过 SSH 执行一个远程主机上的脚本，脚本执行完成后，SSH 连接也将自动中断。

6) SSH 默认的安全机制比起 Telnet 和 RLOGIN 已是足够安全。但是为了满足更高的安全性要求，还需要再做一些额外的设置，所有的配置都在 SSH 服务的主配置文件 `/etc/ssh/sshd_config` 中完成。

❑ 禁止 SSH 使用密码认证连接。设置 `PasswordAuthentication no` 即可，如此一来就杜绝了所有针对密码的暴力破解攻击。

❑ 禁止 root 用户远程登录。设置 `PermitRootLogin no` 即可，我们建议只使用普通用户进行远程登录，并使用 `sudo` 命令来行使大部分 root 权限。如果实在需要使用 root 用户进行交互式管理，可以使用普通用户远程连接到主机，然后通过 `su` 命令切换到 root 用户，这样做更加安全。

❑ 明确指定允许或禁止的远程登录用户。使用 `AllowUsers` 和 `DenyUsers` 来指定哪些用户可以登录，哪些用户不能登录。比如，只允许用户 John 登录，可以设置为：`AllowUsers John`；允许除了 John 以外的其他用户登录，可以设置为：`DenyUsers John`。

❑ 使用非默认端口。SSH 的默认端口为 22，将其改为任意不与其他服务冲突的端口（建议采用 1024 端口以上的端口号）将会使用系统更为安全。在 SSH 配置文件中设置 `Port 2849`，即可修改 SSH 默认端口为 2849。

在 11.2 节之后，我们将对上面这些配置进行运用。

11.1.4 SSH 的发展和远程访问的未来

在过去的十几年间，SSH 一直是远程连接事实上的标准协议。在这期间，整个互联网的

互通性也有了显著改善。SSH 的简单、快速以及安全性，使其无论是在环境可靠且低延迟的局域网还是在环境极其复杂的互联网上都长期受到绝大部分用户的青睐。但是，在网络延迟比较严重的情况下，比如使用 3G、4G 移动网络等，这时使用 SSH 会成为一个非常痛苦的体验。

Mosh (Mobile Shell) 是一个新生的 SSH 的替代品。在高延迟网络环境下，Mosh 要比 SSH 流畅得多。它使用 SSH 建立初始的连接，然后通过 UDP 协议来同步本地 session 和远程主机 session。Mosh 同时更好地支持了 UTF-8 编码，使得其可以被大部分的类 POSIX 操作系统支持，比如 Google Chrome 操作系统。

远程访问技术及工具的未来发展依然充满了各种未知，但有一点是可以确定的，Ansible 将一如既往地提供快速、安全可靠的连接方式，来帮助人们搭建和管理他们的服务器架构。

11.2 通信加密

我们讨论了 SSH 的发展和工作方式，这是因为 SSH 几乎是当今所有基础设施安全的基础。我们允许 SSH 直接连接我们的服务器，所以，了解 SSH 的工作方式和配置方式，同时使用加密通信，对我们的服务器安全至关重要。

下面我们将介绍如何通过 Ansible 对 SSH 进行安全配置。


为了保证服务器的安全性，我们需要禁止 SSH 基于密码登录（禁止密码登录之前，务必确认 SSH 可以成功通过密钥认证登录），使用更为安全的密钥认证来加密通信，禁止 root 用户远程登录，同时改变 SSH 服务的默认端口号。现在 Playbook 配置内容如下：

```
- hosts: example
  tasks:
    - name: 修改 SSH 配置文件的安全选项
      lineinfile:
        dest: /etc/ssh/sshd_config
        regexp: "{{ item.regexp }}"
        line: "{{ item.line }}"
        state: present
      with_items:
        - {
            regexp: "^PasswordAuthentication",
            line: "PasswordAuthentication no"
          }
        - {
            regexp: "^PermitRootLogin",
            line: "PermitRootLogin no"
          }
        - {
            regexp: "^Port",
            line: "Port 2849"
          }
      notify: restart ssh
```



```
handlers:
  - name: restart ssh
    service: name=ssh state=restarted
```

在这个结构十分简单的 Playbook 中，我们使用 Ansible 的 lineinfile 模块对 SSH 的三大安全配置选项进行了设置（不允许密码登录 PasswordAuthentication no，不允许 root 远程登录 PermitRootLogin no，以及修改默认端口为 2849：Port 2849），随后使用 Handler 重启了 SSH 服务，来使修改生效。

 **提示** 如果们改变了某些 Inventory 主机的 SSH 配置，比如修改了默认端口号，此时我们需要在 Ansible 的 Inventory 文件中使用 `ansible_ssh_port` 变量明确地为该主机重新指定新配置的端口号。

11.3 禁止 root 远程登录

在 11.2 节，我们使用了 lineinfile 模块来配置 SSH 完全禁止使用 root 用户远程登录，本节将介绍如何利用 Ansible 更弹性地使用 root 权限。

众所周知，Linux 系统的 sudo 命令可以让普通用户以 root（也可以指定为其他用户）的权限来执行指定命令，这样不仅减少了 root 用户的登录和管理时间，同样也提高了安全性。sudo 命令可以让我们在执行敏感的高权限命令时，更加有针对性，从而减少高权限命令误操作的几率。

Ansible 本身也提倡尽量使用普通用户来远程管理远程主机，只有在必须使用 root 权限的任务中，才使用 sudo 变量来实现 Linux 命令行中的 sudo 功能。比如，当前 Ansible 用户要重启一台 Apache 服务器，但是它没有相关权限，这时就需要在 Playbook 中使用 sudo 关键字来完成操作。

```
- name: Restart Apache.
  service: name=httpd state=restarted
  sudo: yes
```

在任务或 Playbook 中可以通过添加 `sudo_user:[username]` 关键字来指定 sudo 后具体以哪个用户的权限来执行操作，而不仅仅是 root 用户。需要注意的是，`sudo_user` 关键字必须在有 sudo 关键字的前提下才生效。

我们可以使用 Ansible 来编辑 sudo 命令的配置文件来定义哪些用户可以使用哪些命令等与 sudo 有关的详细配置。我们通过下面一个例子来详细了解一下这个过程。

本例中，我们将使用 Ansible 来修改 sudo 的配置文件，使得用户 johndoe 在使用 sudo 命令时拥有和 root 用户一样的权限。

```

- name: 为普通用户赋予所有 root 权限
  lineinfile:
    dest: /etc/sudoers
    regexp: '^%johndoe'
    line: 'johndoe ALL=(ALL) NOPASSWD: ALL'
    state: present

```

当手动对 `sudo` 命令的配置文件进行修改时，我们应当使用修改 `sudo` 配置文件专用的 `visudo` 命令，这能更好地保证对 `sudo` 进行配置，并防止错误修改造成命令的不可用。在使用 Ansible 的 `lineinfile` 模块对 `sudo` 配置文件进行修改时，更应该认真检查，确保修改后其语法的正确性。

另外一种比较稳妥的修改 `sudo` 配置文件方法是，在本地使用 `visudo` 命令对本地 `sudo` 命令的配置文件进行修改，然后使用 `copy` 模块的 `validate` 关键字在将配置文件复制到远程主机之前进行语法检查。代码如下：

```

- name: Copy validated sudoers file into place.
  copy:
    src: sudoers
    dest: /etc/sudoers
    validate: 'visudo -cf %s'

```

上述代码中 `%s` 是一个文件路径的占位符，在文件被复制到远程主机之前，它会被替换为 `src` 后面的文件。

11.4 操作系统简介

在配置文件管理工具流行之前，服务器经常会残留一些不再使用的软件服务，以及这些服务所使用的端口，这不仅使用服务器变得缓慢臃肿，同时这些开放的端口和老旧的软件都给外部攻击造成了潜在风险。

及时关闭服务器上不再需要的服务，卸载不相关的软件，并清理不再需要执行 `crontab` 任务，这不仅可以帮助服务器“瘦身”，还可以提高服务器的安全性。在完全使用 Ansible 来管理维护的服务器架构中，这些工作将变得非常简单。你可使用事先写好的 `Playbook` 或 `Role` 快速地部署一台全新的服务器来取代旧服务器，或者简单地列出一个需要删除的软件列表，利用 Ansible 进行批量卸载。比如下面这个简单实用的例子：

```

- name: 卸载不需要的软件
  apt: name={{ item }} state=absent purge=yes
  with_items:
    - apache2
    - nano
    - mailutils

```

在 Ansible 中，像 `YUM`、`apt`、`file` 以及 `mysql_db` 这些模块，它们都有一个相同的选项

state, 设置其值为 absent, 可以将指定的任务软件、文件或者数据库删除掉。妥善利用这些功能, 可以大大提高运维人员的工作效率, 节省大量时间。

只开放需要用到的端口, 关闭那些可有可无的端口将会大大减少外部环境对主机的攻击面, 同时也会降低我们防火墙的复杂度。举例来说, 不加任何限制就对外开放的 25 号端口会给外部网络提供大量的主机信息, 所以, 如果你的主机不是一台 SMTP 服务器的话, 务必关闭这个端口。同时, 也要确保那些需要被开启的端口, 只能连接依赖的客户端。

11.5 遵守权限最小化原则

生产环境中, 主机上面所有的用户、应用以及进程都应该只允许访问他们本身需要访问的信息(文件)和资源(内存、网络商品等), 一点也不多, 一点也不少。

本章节中介绍的其他几种安全策略都涉及权限最小化原则。在权限最小化原则实施的过程中, 最直接也是最基本的两个方向就是: 用户权限管理与文件权限管理。二者看似简单, 实则与整个系统的安全息息相关。

11.5.1 用户管理

系统上的每一个新增用户的权限默认都是被适当限制过的。新增用户通常都有一个家目录, 且用户对家目录下的所有文件和目录具有最高权限, 但是对于家目录以外的目录与文件的权限都需要被重新赋予。

通常, 为用户新增权限的方法有两种。

□ 添加用户至其他用户组中, 以继承该用户组的权限。

□ 为用户开放 sudo 权限, 使得其可以以 root 或其他用户的身份来执行命令或访问文件。

以上这些就是用户权限管理的基本方法, 更多方法可以参考 11.5.2 节的文件权限管理方法。

11.5.2 文件权限管理

Ansible 中每一个与文件管理相关的模块中都有文件权限管理的选项可用, 这些选项包括: owner、group 和 mode。每一次当我们使用 copy、template、file 等模块来操作管理文件的时候, 都应该使用这些选项来明确指定文件权限及其归属。比如, Gitlab 的配置文件应该只能被 root 用户读取和修改, 其他任何用户都没有权限。我们可以进行如下配置:

- name: 设置 Gitlab 配置文件的权限

file:

path: /etc/gitlab/gitlab.rb

owner: root

group: root

mode: 0600

对于一些新手管理员来说，经常会给文件设置过于宽松的权限，比如当服务遇到文件权限问题的时候，为了方便省事，直接把相关文件权限设置为 777，甚至对整个目录设置 777 的权限。

为了满足用户对某些文件或目录的权限需求，比如 Web 服务器上的 httpd 用户或 Nginx 用户需对网站文件拥有权限，正确的做法是修改文件或目录的权限来适应用户，而不是扩大用户的权限来得到某些权限的满足。

11.6 定期维护更新

在我们的服务器上，每一年所有软件的安全更新有上百次甚至更多。这其中有一些是针对修复对系统有严重威胁的漏洞的（比如当年的 Heartbleed 漏洞），如果这些漏洞没有及时更新软件或打相应的补丁，将会对系统安全造成严重威胁。

我们应该安排定期进行补丁维护和软件更新检查。在对线上生产服务器上的软件打补丁或更新升级之前，应当在非关键服务器或环境相同的测试服务器上进行测试，在确定没问题的情况下再对线上生产服务器进行操作。

11.6.1 手动更新

在使用了 Ansible 维护的服务器架构中，我们只需要下面一条简单的 Ansible 命令就可以完成系统上所有软件的更新升级操作。

对于 RedHat 和 CentOS 等系统来说，使用如下命令：

```
ansible webservers -m yum -a "name=* state=latest"
```

对于 Debian 和 Ubuntu 等系统来说，使用如下命令：

```
ansible webservers -m apt -a "upgrade=dist update_cache=yes"
```

然而，在有些情况下我们只需实施与安全有关的更新，或者只更新某些软件。在仍需要使用这两个命令的前提下，我们可以通过修改 YUM 软件和 apt 软件的配置文件来进行定义。

11.6.2 自动定时更新

我们还可以在系统上设置每天或每周定时进行软件更新，这样可以使系统更新这一动作更稳定、更有规律地执行下去，从而减少人力成本。但是现实中，有些环境是不允许机器自动更新软件的，因为自动更新本身蕴含着一些风险，比如有些软件的最新版确实修正了之前版本的一些不足，但是它的新增功能可能与系统上自己开发的一些程序的兼容性不足，从而使得整个系统不可用。如果你的系统上不存在这些问题，那么使用自动定时更新软件，可以更进一步增加系统安全性。

1. 自动更新 RedHat 系统上的软件

对于 RedHat 6 及其以后版本的系统来说（包括 Fedora 和 CentOS）都可以使用一个叫 YUM-cron 的软件进行软件包更新管理。其用法很简单，使用 YUM 安全后，保证开机启动就可以了。使用 Ansible 来实现如下即可：

```
- name: 安装 yum-cron.
  yum: name=yum-cron state=present

- name: 运行 yum-cron 并设置开机启动
  service: name=yum-cron state=started enabled=yes
```

更多的配置可以通过修改 YUM 的配置文件 /etc/yum.conf 来实现。

2. 自动更新 Debian 系统上的软件

Debian 系统及其衍生版都使用一款名叫 unattended-upgrades 的软件来实现自动化软件包更新管理，这款软件和前面讲的 YUM-cron 一样，非常便于安装和配置，并且支持多配置文件，存放于 /etc/apt/apt.conf.d/。

```
- name: 安装 unattended-upgrades
  apt: name=unattended-upgrades state=present

- name: 将配置文件复制到配置目录中
  template:
    src: "../templates/{{ item }}.j2"
    dest: "/etc/apt/apt.conf.d/{{ item }}"
    owner: root
    group: root
    mode: 0644
  with_items:
    - 10periodic
    - 50unattended-upgrades
```

上例中复制的 unattended-upgrade 配置文件 10periodic 的内容如下：

```
# File: /etc/apt/apt.conf.d/10periodic
APT::Periodic::Update-Package-Lists "1"; // 显示更新包列表，0 表示停用设置
APT::Periodic::Download-Upgradeable-Packages "1"; // 下载更新包，0 表示停用设置
APT::Periodic::AutocleanInterval "7"; // 7 天自动删除
APT::Periodic::Unattended-Upgrade "1"; // 启用自动更新，0 表示停用自动更新
```

配置文件 50unattended-upgrades 的内容如下：

```
# File: /etc/apt/apt.conf.d/50unattended-upgrades
Unattended-Upgrade::Automatic-Reboot "false";

Unattended-Upgrade::Allowed-Origins {
    "Ubuntu lucid-security";
    // "Ubuntu lucid-updates";
};
```

这个配置文件提供了更新配置选项，比如对于那些更新后需要重启服务器才能生效的软件，在更新过这些软件后，是否自动重启服务器，以及在检查更新软件，需要检测哪些 APT 源来查找更新等。

11.7 善用 Iptables 防火墙

假如现在我们建设一所银行的金库，我们肯定不想这座金库有太多的门和窗户与外界相通。相反我们使用更坚固的墙，以及最多一到两个结实的强金属大门。

同样的道理，对于我们的服务器而言，那些没有明确指明需要开启的端口就像金库中多余的窗户一样，需要被完全关闭。对于那些需要被开启的端口，也需要对它的访问权限进行有效的限制。这就在防火墙的职责范围内。如今已经有好多种管理防火墙的工具，比如 iptables、ufw 以及 firewalld 等。另外一些云主机供应商也提供了一些针对自身平台主机的防火墙服务，比如 AWS 的安全组以及阿里云的安骑士等。

在 Debian 及衍生系统中，我们使用管理防火墙。下面一个例子中，我们将看到如何借助 Ansible 来关闭 Debian 系统中除了 22 (SSH)、80 (HTTP)、123 (NTP) 号端口以外的其他所有端口。

```
- name: 使用 ufw 模块来管理哪些端口需要开启
  ufw:
    rule: "{{ item.rule }}"
    port: "{{ item.port }}"
    proto: "{{ item.proto }}"
  with_items:
    - { rule: 'allow', port: 22, proto: 'tcp' }
    - { rule: 'allow', port: 80, proto: 'tcp' }
    - { rule: 'allow', port: 123, proto: 'udp' }

- name: 配置网络进出方向的默认规则
  ufw:
    direction: "{{ item.direction }}"
    policy: "{{ item.policy }}"
    state: enabled
  with_items:
    - { direction: outgoing, policy: allow }
    - { direction: incoming, policy: deny }
```

上述 Playbook 任务运行之后，登录对方主机，使用 `sudo ufw status verbose` 命令可以看到详细的配置信息。

```
Status: active
Logging: on (low)
Default: deny (incoming), allow (outgoing), disabled (routed)
New profiles: skip
```

```

To Action From
-----
22/tcp ALLOW IN Anywhere
80/tcp ALLOW IN Anywhere
123/udp ALLOW IN Anywhere
22/tcp (v6) ALLOW IN Anywhere (v6)
80/tcp (v6) ALLOW IN Anywhere (v6)
123/udp (v6) ALLOW IN Anywhere (v6)


```

在 Redhat 及其衍生系统中，我们使用 Ansible 的 `firewalld` 模块来完成防火墙的管理。

```

- name: 使用 firewalld 模块管理端口
  firewalld:
    state: "{{ item.state }}"
    port: "{{ item.port }}"
    zone: external
    immediate: yes
    permanent: yes
  with_items:
    - { state: 'enabled', port: '22/tcp' }
    - { state: 'enabled', port: '80/tcp' }
    - { state: 'enabled', port: '123/udp' }

```

 **注意** 本例中，`immediate` 选项从 Ansible 版本 1.9 之后开始引入，用来定义规则在配置完成后是否立即生效。如果使用的是 1.9 之前的版本的，那么需要重启防火墙来让新规则生效，或者将 `permanent` 选项的值设为 `no`。

`firewalld` 模块并不能针对网络进出口方向进行管理，但是我们可以借助 `iptables` 模块或者直接修改 `/etc/firewalld` 目录下的防火墙配置文件来进行配置。

我们使用 `firewall-cm` 命令可以查看当前系统上哪些端口是被防火墙放行的。

```
sudo firewall-cmd --zone=external --list-all
```

运行结果如下：

```

external
  interfaces:
  sources:
  services: ssh
  ports: 123/udp 80/tcp 22/tcp
  masquerade: yes
  forward-ports:
  icmp-blocks:
  rich rules

```

最后，使用什么方法来管理防火墙并没有太大区别，重要的是要遵循权限最小化原则，只开放端口给那些需要依赖此端口才能正常实现其功能的主机。



注意 在配置防火墙时，一定要注意的一点是：不要一不小心把自己管理主机要用到的 IP 地址和端口给屏蔽了，这样，你将不得不亲自或请人直接连接服务器的本地终端进行解封。

11.8 定期磁盘巡检

通过检查服务器日志，不仅可以查看已经发生的攻击情况，还可以通过已有日志记录来预测高流量攻击和潜在风险。但是，如果日志没有被正确地管理和分析，那么它将变得毫无用，甚至会对系统造成拖累，占用大量磁盘空间，影响磁盘性能。

不同的服务器上需要监控和管理的日志种类也各不相同，但是通常来说，需要多加注意的日志种类有：数据库的慢查询日志、Web 服务器的访问日志和错误日志、系统认证审计日志以及 crontab 日志。我们可以采用目前比较流行的 ELK 系统对日志进行全面的分析，也可以利用 Nagios、Zabbix 等监控工具对日志进行监控报警。

此外，我们还应该使用类似 logrotate 的日志管理工具对日志文件进行定期的切割和归档，当然可以根据自身需要编写脚本，来实现更符合自身业务场景的日志管理工具。我们应该对日志文件的大小多加留意，谨防其尺寸太大，占用过多磁盘空间而影响其应用的正常运行。大日志切割归档时，可以根据日志文件的大小来制定规则，并及时清除过期的日志文件。

11.9 系统登录日志审计

任何一台面向公网开放 22 号端口并允许使用密码远程登录的主机，都无时无刻不面临着洪水般的密码暴力三角攻击。对于很多知名的服务器站点，通常情况下每小时都可以侦测到上万次的攻击。

如果在不得已的情况下，允许用户使用密码远程登录你的主机，那么这时就需要部署一些针对登录情况的监控机制，以及设置登录频率的限制。至少需要安装像 DenyHosts 和 Fail2Ban 这类可以根据日志文件、基于登录失败频率来自动屏蔽可疑 IP 的软件。

下面介绍如何通过 Ansible 同时在 Debian 和 RedHat 系统上批量部署 Fail2Ban 软件，并设置开机启动。

```
1 - name: 在 RedHat 系统上安装 Fail2Ban
2 yum: name=fail2ban state=present enablerepo=epel
3 when: ansible_os_family == 'RedHat'
4
5 - name: 在 Debian 系统上安装 Fail2Ban
```



```

6 apt: name=fail2ban state=present
7 when: ansible_os_family == 'Debian'
8
9 - name: 启动软件，并设置开机启动
10 service: name=fail2ban state=started enabled=yes

```

Fail2Ban 软件的配置文件由 /etc/fail2ban 目录下的多个以 .conf 结尾的文件构成，这些文件中的配置可以覆盖默认 /etc/fail2ban/jail.local 中的配置。

11.10 正确使用 SELinux 和 AppArmor

SELinux 和 AppArmor 都可以用来为内存和文件系统构建安全沙盒，比如控制一个应用不能访问另一个应用的资源，这有点像文件的属主和属组权限的管理，但是与之相比，SELinux 和 AppArmor 能做到更为精细和复杂的权限控制。但正是由于它们过于详细的权限控制和复杂配置，大多数人并不愿意开启这两款软件。尤其是对于安装了一些非大众软件的服务器，这两款软件对它的支持并不好，但是对于像 Apache 和 MySQL 这样的大众软件，这两款软件对它们的支持还是非常好的。

相对于 AppArmor 来说，SELinux 的流行度更高一些。SELinux 的预设策略有两种，其一是 strict（完全限制的 SELinux 保护），另外一种是 targeted（仅对网络服务限制严格的 SELinux）。如果你的服务器上运行了多个服务，并且包含了面向 Web 的应用，那么这种场景下使用 SELinux 的 targeted 策略将会在各应用间提供更好的安全隔离。

要合适 Ansible 配置 SELinux 开启 targeted 策略，需要安装 Python 的 SELinux 库，然后利用 Ansible 的 selinux 模块开启 targeted 策略。

```

- name: Install Python SELinux library.
  yum: name=libselinux-python state=present
- Ensure SELinux is enabled in `targeted` mode.
  selinux: policy=targeted state=enforcing

```

还可以利用 Ansible 的 seboolean 模块来设置 SELinux 的 booleans。最常见的用法是通过修改 httpd_can_network_connect 的 boolean 值来为 Web 服务开通访问外网的权限。代码如下：

```

- name: 为 Web 服务开通访问外网的权限
  seboolean: name=httpd_can_network_connect state=yes persistent=yes

```

Ansible 的 file 模块与 SELinux 也有非常好结合，这使得 file 模块可以为文件和目录设置 4 种安全上下文，分别为：selevel、serole、setype、euser。它们分别作用于不同的对象和范围。

SELinux 除了 strict 和 targeted 两种默认策略外，还支持自定义策略，但是定义方法极其复杂，也超越了本章的介绍范围，这里将针对这部分进行深入讲解。但是我们应该能够熟悉使用像 setroubleshoot、etroubleshoot-server、getsebool 以及 aureport 这样的命令来查看哪些

内容是用 SELinux 屏蔽的, 哪些 boolean 是可用的, 以及哪些 boolean 是开启的、哪些是关闭的。

下一次, 当你准备关闭一台新服务器的 SELinux 的时候, 要先稍微花些时间思考一下, 再根据自身的业务场景来设置正确的 SELinux 权限。小小的习惯改变, 将会对服务器安全带来质的提升。

11.11 本章小结

本章节是对 Linux 系统安全实践的一个概览, 同时展示了如何借助 Ansible 来实现这些安全措施。系统安全无小事, 每一个安全细节的完善对整个系统来说都有巨大意义, Ansible 在这方面也有着非常完美的支持。善用 Ansible 不仅可提高我们的运维工作效率, 也能提升系统的整体安全性。

我们只要到 Ansible 的目录 `core` 中找到 `ansible-core` 模块包，并将其复制到 `~/.ansible/modules/core`。

第三篇 *Part 3*

Web 自动化开发篇

- 第 12 章 Ansible 模块编写
- 第 13 章 开发自己的 Ansible WebUI
- 第 14 章 Web 与 Ansible 结合的常用实例

Ansible 默认安装目录是 `/usr/share/ansible/`，在该目录下新建文件名为 `ad-hoc.yml`，并

内容是用 SELinux 来保护的。有些 bookcan 是可用的，以及哪些 bookcan 是开启的。哪些是关闭的。

下一次，我们将看到 SELinux 是如何工作的。在配置 SELinux 的时候，要先稍微花些时间思考一下，再根据自身的需要进行配置。

再根据自身的需要进行配置。SELinux 的配置，小小的习惯改变，将会对服务器安全产生深远的影响。

Chapter 12

第 12 章

Ansible 模块编写

本章是对 Linux 系统安全实践的一个概述，同时展示了如何借助 Ansible 来管理 Linux 系统。本章将介绍 Ansible 的基本概念，包括 Ansible 的架构、安装、配置、使用等。本章还将介绍 Ansible 的模块编写，包括模块的编写、测试、部署等。本章还将介绍 Ansible 的进阶应用，包括 Ansible 的集成、扩展、定制等。本章还将介绍 Ansible 的社区资源，包括 Ansible 的文档、论坛、邮件列表等。

如今 Ansible 在 GitHub 上的关注度在系统管理领域已经跃居第一，这不仅仅归功于它易于使用、便于部署的特点，有一点我们不能忽略，那就是它有众多的项目贡献者，无论是基于核心代码还是模块，都有开发者加入其中。Ansible 已经为大家提供了模块编写基础库，我们只要学会使用它，就可以开始 Ansible 模块的编写，从而为该项目贡献自己的一份力量。Ansible 模块的编写其实并不困难，通过本章内容的学习，你就可以了解目前大多数一直在被我们使用的 Ansible 模块的工作原理。

12.1 初步认识 Ansible 模块

在写 Ansible 模块之前，需要了解我们现在使用的一些常用模块的工作原理。我们将分如下 3 个方面介绍：

- 如何使用这些模块；
- 程序为什么能识别它们；
- 模块的工作特性。

(1) 如何使用这些模块

如果要使用系统的 Ansible 模块，在命令行中，我们只要指定 `-m` 后添加这个模块名就可以了（例如，我们平时使用的 `command` 和 `shell` 均属于系统模块。执行命令 `ansible-doc list` 获取所有系统模块），而在 Ansible-playbook 中，我们在编写 YAML 格式的文件时，只要把模块名写在该行开头并加上冒号，就可以继续使用了（在 YAML 文件某行写上我们经常使用的模块，如 `shell: echo xxx`）。

(2) 程序为什么能识别它们

有两种方法可以使 Ansible 程序去识别模块。

1) 寻找 Ansible 代码的模块所在目录。我们只要到 Ansible 代码目录下找到 core 文件夹 (比如 /usr/local/lib/python2.7/dist-packages/ansible/modules/core), 把我们编写的模块放到其中, 该模块即可被我们调用了。

2) 通过 Ansible 提供的配置文件制定模块的目录。我们可以把自己编写的模块放到 /etc/ansible/ansible.cfg 中 library 定义的目录下 (比如 library = /mywork/ansible/modules/), 这种方法比较推荐。

以上两种方法都可以使我们之前编写的模块正常被 Ansible 程序所识别。

(3) 模块的工作特性

所有 core 目录下的 Ansible 模块, 如果使用到 Ansible 提供的 ansible.module_utils.basic 这个基础模组, 所有在模块里写的程序都是到远程机器才执行的, 所有操作将不再和安装有 Ansible 的主机有联系了。可能读者会问, 比如 ping 这个模块, 以及一些传输的模块 (copy、synchronize 等), 它们确实在 core 目录下面, 但好像都与 Ansible 的主机有关系。这里解释一下, 关于 ping 这个模块, 相当于登录远程机器上 echo 了一个 pong, 而 copy 和 synchronize 这些文件传输模块不是用的 core 目录下的这些模块, 而是使用了 unner/action_plugins 目录下的对应模块。所以我们在编写 Ansible 模块的时候, 一定要注意, 我们一旦使用了 ansible.module_utils.basic 这个基础模组编写出来的模块将不能对装有 Ansible 的主机做任何操作, 所有操作都是在 agent 的机器上面的。

12.2 Ansible 简单模块编写

接下来, 我们来编写一个简单的 Ansible 模块。先看下我们要实现的目标命令:

```
ansible xxx -m echopong
```

我们在远程机器 xxx 上执行以下命令 (比如 echo pong):

```
ansible xxx -m shell -a "echo pong"
```

那么我们就来完成和以上命令性质相同的模块吧。

在开始编写属于我们自己的模块之前, 我们要做几个准备工作。

步骤 1: 创建一台测试的 agent 机器 test。

在把 test 机器与我们的 Ansible 主机建立信任后, 再在 Ansible 的 hosts 配置文件上填写对应的信息 (比如 test xxx.xxx.xxx.xxx) 即可。

步骤 2: 在对应目录下新建模块 echopong。

如 12.1 节介绍, 要让程序去识别我们的模块, 就必须先定义一个目录, 这里我们就使用 Ansible 默认模块目录 /usr/share/ansible/, 在该目录里创建文件名为 echopong 的模块。

步骤 3: 开始设定我们的目标, 并开始编写。

同样, 我们要在远程机器 `test` 上执行 `echo pong`, 但不使用 `Shell` 模块来实现 `echo pong`, 实现的效果如以下命令:

```
ansible test -m echopong
```

为了实现这个目的, 我们用图 12-1 `echopong` 模块编写流程图表示, 绝大多数 Ansible 模块的编写都依据此流程。在图 12-1 流程图的第 4 步, 大家可以更换其他的执行程序大纲, 还可以在第 4 步增加退出模块的相关条件, 从而对模块进行横向扩展。

1) 为该模块取名为 `echopong`, 并放入 `/usr/share/ansible/`, 文件在本章的 `12_2/echopong`。

2) 第 1 行写上路径声明的话(不写的话, 运行的时候可能会报错), 第 2 行则是引入一些基础的 Ansible 模块组件。具体示例如下:

```
# !/usr/bin/python
from ansible.module_utils.basic import *
import os
```

3) 为模块传入必要的参数, 因为在这里我们不需要传递参数, 那么在参数方面, 我们只要简单地调用 Ansible 默认传递参数的类就可以了。

```
module = AnsibleModule(argument_spec = dict())
```

4) 执行我们要执行的 `Shell` 命令。

```
os.system('echo pong')
```

5) 提供结果信息 `echo pong`, 主要为了告诉我们这个模块干了什么。

```
result = dict(echo='pong')
```

6) 退出 Ansible 这次模块, 并返回信息。

```
module.exit_json(**result)
```

你可以下载这些代码段, 我们会把代码放在 `GitHub` 上。

```
cd your_module_path # cd 到你已经定义的模块路径, 如 /usr/share/ansible/
wget https://raw.githubusercontent.com/stanley1st/ansibleUI/master/Chapter_12/12-2/echopong
```

这样, 我们写了这短短的 6 行就可以实现在远端机器完成 `Shell` 命令 `echo pong` 这个操作, 读者只要运行 `ansible test -m echopong` 这条命令进行测试即可。



图 12-1 `echopong` 模块编写流程图

12.3 模块变量添加

通过 12.2 的内容，我们成功地完成了 echo pong 这个模块的编写。但仅仅这样的话，这个模块就显得太单薄了，没法给使用者更多的选择，如果能把变量引入其中的话我们就可以像脚本一样去传入参数了。那么我们接下来就要学习怎么传递参数到模块中了。我们先设定一个目标，把 echo pong 改成 echo 我们传递进去的一个参数。假定以下命令：

```
ansible test -m echo -a "args='pongpong'"
```

首先我们需要在图 12-1 流程图中第 3 步传递一个叫 args 的参数：

```
module = AnsibleModule(
    argument_spec = dict(
        args=dict(required=True)),
)
```

required=True 说明这条命令必须要有“args='xxx'”，否则将会报错。

对应地，我们需要修改第 4 步中的 shell 操作，把这个 args 的变量传递到 Shell 中去，这样我们就成功地把我们定义的参数 args 引入到这个模块中去了。

```
args = module.params['args']
os.system('echo {0}'.format(args))
```

最后，我们只要修改第 5 步中的输出结果就可以了。

```
result = dict(echo=args)
```

其他代码不变，让我们来测试下，当运行 `ansible test -m echo -a "args='pongpong'"` 这条命令之后结果如下：

```
test | SUCCESS => {
    "changed": false,
    "echo": "pongpong"
}
```

这样我们就完成了传递参数的过程。只要学会这些，我们就可以写一些自定义的基础 Ansible 模块了。

同样，为了节省大家时间，可以下载 GitHub 上的代码片段。

```
cd your_module_path
wget https://raw.githubusercontent.com/stanleyst/ansibleUI/master/Chapter_12/12-3/echopong
```

我们还可以定义其他变量的类型，下列都是我们经常使用的变量类型：

必填项：name= dict(required=True)。

默认值：default= dict(default='present')。

选择项：choices= dict(default='present', choices=['present', 'absent'])。

布尔值: `bools= dict(type='bool')`。

字符型: `str= dict(type='str')`。

任选变量: `name1= dict(aliases=['name2', 'name3'])`。

注: 任选变量的用法为变量引用时, 使用 `name1`、`name2`、`name3` 这三个变量是一样的 (例如: 执行命令 `ansible test -m echo -a "name1=' pongpong '"`)

等价于

`ansible test -m echo -a "name2=' pongpong '"`

等价于

`ansible test -m echo -a "name3=' pongpong '"`。

我们只要修改第 4 步中的变量的赋值类型就可以更换我们模块中所需要的变量的类型了。大家可以自己尝试填入下面的模板:

```
module = AnsibleModule(
    argument_spec = dict(
        Args1= 自定义区域 1,      # Args1 = dict(default='present')
        Args2= 自定义区域 2,      # Args2 = dict(type='bool')
    )
)
```

12.4 模块状态返回的标识及应用

当我们执行自己编写的模块 `echopong` 的时候, 会得到以下返回值:

```
test | SUCCESS => {
    "changed": false,
    "echo": "pongpong"
}
```

其中有一个我们并没有定义的返回值 `"changed": false`, 如果我们不做调整的话, 结果中每次都会返回一个 `"changed": false`。针对这个问题, 我们要解决 3 个问题:

- 1) `"changed": false` 到底代表什么含义;
- 2) 可不可以把它改成 `"changed": true`;
- 3) 是不是可以任意添加其他返回值的结果。

第 1 个问题: 其实 `changed` 并不代表什么, 它只是代表一个记录的值, 我们后续可以根据这个值做一些逻辑判断。它的好处在于不像返回值中 `failed` 那么严重, 当 `"failed": True` 时, 会导致我们使用 `ansible-playbook` 时, 在该出错模块上停止工作。

第 2 个问题: 那我们怎么把它改为 `"changed": true` 呢? 其实很简单, 我们简单地吧图 12-1 流程图中第 5 步的 `changed=True` 赋给返回的结果即可。

```
result = dict(echo=args, changed=True)
```

那么我们得到的返回值的结果如下:


```
test | SUCCESS => {
    "changed": true,
    "echo": "pongpong"
}

cd your_module_path
wget https://raw.githubusercontent.com/stanleyst/ansibleUI/master/Chapter_12/12-4/1/echopong
```

第 3 个问题：根据上述的推算，我们就将引出很多返回的值，可以任意更改结构。比如我们可以把图 12-1 流程图中的第 5 步修改成这样：

```
result = dict(echo=args, changed=True, good='good', bad='bad')
```

那么我们得到的返回值的结构将是：

```
test | SUCCESS => {
    "bad": "bad",
    "changed": true,
    "echo": "pongpong",
    "good": "good"
}

cd your_module_path
wget https://raw.githubusercontent.com/stanleyst/ansibleUI/master/Chapter_12/12-4/2/echopong
```

但值得提醒一点的是，我们必须避免使用 Ansible 模块中已经被使用的一些字段，它们的意义是不一样的。比如当你使用了这些字段，那就代表执行的结果是错误的，会显示红色字样，这点必须注意。

```
result = dict(echo=args, changed=True, failed=True, rc=0)
```

同样，读者可以试试把 `rc` 这个值不赋值为 0，看看返回结果又是什么。

```
cd your_module_path
wget https://raw.githubusercontent.com/stanleyst/ansibleUI/master/Chapter_12/12-4/3/echopong
```

我们还应该注意的是，在返回值中应该避免重定义系统模块变量，这很重要。下面我们来举个例子，比如 `invocation` 我们也不能使用，因为无论我们怎么声明，到最后它都会被覆盖掉。那我们来试试吧。

```
result = dict(echo=args, invocation=0)
```

当然，我们期待的结果应该如下：

```
test | success >> {
    "changed": true,
    "invocation": 0,
```

```
    "result": "pongpong"
}
```

但其结果却如下:

```
test | success >> {
  "changed": true,
  "result": "pongpong"
}
```

这是因为 `invocation` 这个值是 `ansible-playbook` 的关键字, 我们可以猜测到, 在最后赋值后, 被程序再一次赋值了, 冲掉我们所给的值。从上述情况看来, `invocation` 应该是一个空值, 我们可以进一步去验证它。我们使用这个模块写一个 `ansible-playbook`, 来测试下这个值是否在 `playbook` 的结果中出现, 结果是否会被我们定义的值所覆盖。我们定一个文件名为 `echopong.yml`。

```
- name: echopong
  hosts: localhost          # 加上 /etc/ansible/hosts
                             # localhost ansible_connection=local
  tasks:

    - name: echo pongpong
      echo: args="pongpong"
      register: res
    - debug: var=res
```

接下来, 我们执行命令:

```
ansible-playbook echopong.yml
```

我们可以得到的结果如下:

```
... ..
TASK: [echo pongpong] *****
changed: [test]

TASK: [debug var=res] *****
ok: [test] => {
  "var": {
    "res": {
      "changed": true,
      "invocation": {
        "module_args": "args=\"pongpong\"",
        "module_complex_args": {},
        "module_name": "echo"
      },
      "echo": "pongpong"
    }
  }
}
```

获取代码:

```
wget https://raw.githubusercontent.com/stanley1st/ansibleUI/master/
Chapter_12/12-4/4/echopong.yml
cd your_module_path
wget https://raw.githubusercontent.com/stanley1st/ansibleUI/master/
Chapter_12/12-4/4/echopong
```

返回的结果中 `invocation` 这个值不是我们设定的 1，仍然为它原来的值。通过上述例子说明，我们要尽量避免使用 Ansible 输出结果的一些关键字。

其实我们定义额外值的目的在于，可以为 Ansible-playbook 中的条件提供判断，这样这个模块就更具有复杂性和实用意义了。

那我们就来简单尝试一下。

这里我们增加一个值 `pass_code`，文件 `echopong.py` 如下：

```
result = dict(echo=args, pass_code=0)
```

接下来，Ansible-playbook 就可以使用这个值来做判断了。这样，可以让我们的 Ansible-playbook 更具有可控性。我们来编写一个 `echopong.yml` 的例子来使用 `pass_code`，当 `pass_code` 的值为 1 的时候，我们输出一个正确值。

```
- name: echopong
  hosts: localhost
  tasks:

    - name: echo pongpong
      echo: args="pongpong"
      register: res

    - name: check_code
      debug: msg=ok
      when: res.pass_code == 1
```

相信大家已经得到自己所要的结果了！这里就不再列出来了。

可使用 GitHub 上所提供的示例。

```
wget https://raw.githubusercontent.com/stanley1st/ansibleUI/master/
Chapter_12/12-4/5/echopong.yml
cd your_module_path
wget https://raw.githubusercontent.com/stanley1st/ansibleUI/master/
Chapter_12/12-4/5/echopong
```

通过本节，我们理解了 Ansible 模块的返回值的一些用法和注意点，这能够更好地帮助我们灵活运用这些返回值，为后续写更复杂的 YML 文件的判断做准备。

12.5 模块退出状态处理

为了让模块返回我们所要的结果，来展示正确或错误，Ansible 模块的编写很有必要定

义模块的退出。退出无非两种情况：正常退出；错误退出。在 12.3 节中，我们使用了正常退出的函数 `api: module.exit_json`，对应的还有一个错误退出的 `api: module.fail_json`。但要注意的是，它所需要的传递不再是任意参数了，而是 `msg`，所以它的写法应该是：

```
module.fail_json(msg="errors happened")
```

这样的话，我们就可以配合条件判断，来组织我们自己的模块退出的形式了。这里我们假定有一个 `rc` 的变量为 0 是正常退出，反之则错误，文件名为 `10_5_echopong_with_vars_return_quit.py`。

```
# ... .. 上述省略
if rc == 0:
    result = dict(echo=args, changed=True)
    module.exit_json(**result)
else:
    module.fail_json(msg="errors happened")
```

`module.exit_json` 和 `module.fail_json` 最大的区别在于，在执行完该模块后，`module.fail_json` 将导致整个 Ansible-playbook 停止工作，可以看做一个异常抛出来处理。

```
cd your_module_path
wget https://raw.githubusercontent.com/stanleyst/ansibleUI/master/
Chapter_12/12-5/echopong
```

12.6 模块其他功能补充

对于 `supports_check_mode` 这个属性的使用，如果在执行时使用 `--check`，Ansible 才会去检查该模块是否可用，我们一般可以忽略。当你设定 `supports_check_mode=True` 时，就意味着该模块已经经过自己的严格测试了，可以提交给别人使用了，而且你必须对这个模块负责。但我们自己写的模块一般都是自己用，所以不必理会这个参数。

```
module = AnsibleModule(
    argument_spec = dict(...),
    supports_check_mode=True
)
```

如果我们没加上 `supports_check_mode=True` 这个参数，那么当我们再加上 `--check` 时，我们写的这个模块将自动被忽略。

```
wget https://raw.githubusercontent.com/stanleyst/ansibleUI/master/
Chapter_12/12-4/5/echopong.yml
cd your_module_path
wget https://raw.githubusercontent.com/stanleyst/ansibleUI/master/
Chapter_12/12-6/5/echopong
```

执行以下 `playbook`：


```
ansible-playbook echopong.yml --check
```

其结果将为：

```
... ..
TASK: [echo pongpong] *****
skipping: [test]
ok: [test]
```

```
TASK: [debug var=res] *****
ok: [test] => {
  "var": {
    "res": {
      "changed": false,
      "invocation": {
        "module_args": "args=\"pongpong\"",
        "module_complex_args": {},
        "module_name": "echo"
      },
      "msg": "remote module does not support check mode",
      "skipped": true
    }
  }
}
... ..
```

可以看到，该模块在上面已被 skipping 了 (skipping: [test])。

现在，读者们可以去阅读 /usr/share/ansible 目录下的一些模块了，只要认真读完这些内容，大多数模块都是比较简单的，而且大多数都是使用拼接 shell 命令的方式完成的，只不过加上了一些逻辑判断而已。现在看，其实 Ansible 的模块并不是那么神奇，我们完全可以加人写自己的模块。

12.7 Ansible 模块 API 的调用

由于 Ansible 版本上有所区别，如果读者使用的是 Ansible 的 1.9.x 之前的版本，它的 API 调用就相对比较简单，但缺点也很明显，每个模块都要重新去连接一次远程的 Agent 机器，就不像写 Ansible-playbook 那样，可以连接一次后，执行多个模块。那么我们就把前几节编写的 echopong 嵌入我们的 API 中去。

(1) Ansible 1.9.x 的 API 调用方法

```
import ansible.runner
runner = ansible.runner.Runner(
    module_name='echopong',
    module_args="args='ok'",
    pattern='localhost',
```

```
forks=10
)
datastructure = runner.run()
print datastructure
```

在上面我们写的脚本中，使用到了 `ansible.runner` 这个 Ansible 的 API，我们简单地执行它，就可以得到我们所需要的结果。

```
{'dark': {}, 'contacted': {'test': {'invocation': {'module_name': 'echo',
'module_complex_args': {}, 'module_args': u'args='ok'"}, u'changed': True,
u'pass_code': 1, u'result': u'ok'}}}
```

```
wget https://raw.githubusercontent.com/stanleyst/ansibleUI/master/
Chapter_12/12-7/tag_1.9.x/old_api.py
```

(2) Ansible 1.9.x 的 API 中 Runner 可以更改的参数

因为结果输出是很标准的字典，接下来我们就可以取所需要的值进行输出了（例如：`print datastructure['contacted']`），我们就可以取到自己最想要的结果。

看下这个 Runner 函数可以传入哪些参数呢？

```
host_list=C.DEFAULT_HOST_LIST, # ex: 指定 ansible 的 hosts 文件 /etc/ansible/hosts
module_path=None, # ex: 指定模块目录 /usr/share/ansible
module_name=C.DEFAULT_MODULE_NAME, # ex: 指定模块 (copy)
module_args=C.DEFAULT_MODULE_ARGS, # ex: 指定参数 ("src=/tmp/a dest=/tmp/b")
forks=C.DEFAULT_FORKS, # 指定开启多少线程
timeout=C.DEFAULT_TIMEOUT, # 设定超时时间
pattern=C.DEFAULT_PATTERN, # 指定 hosts 中的机器或机器组
remote_user=C.DEFAULT_REMOTE_USER, # ex: 远程用户名
remote_pass=C.DEFAULT_REMOTE_PASS, # ex: 远程密码，没有则不设定
remote_port=None, # 如果不是 22 端口，请指定
private_key_file=C.DEFAULT_PRIVATE_KEY_FILE, # 指定信任文件
background=0, # 指定每 xx 秒同步，0 则为不同步
basedir=None, # 指定 ansibleplaybook 目录
setup_cache=None, # 用来共享 fact 数据或其他任务
vars_cache=None, # 用来储存关于 host 的变量
transport=C.DEFAULT_TRANSPORT, # 选择使用 'ssh', 'paramiko', 'local' 之一
conditional='True', # 当该 fact 表达式是 true 时，才运作
callbacks=None, # 用于输出
module_vars=None, # Ansible-playbook 内部变量
play_vars=None,
play_file_vars=None,
role_vars=None,
role_params=None,
default_vars=None,
extra_vars=None, # 定义传入 Ansible-playbook 的变量
is_playbook=False, # 是否使用 Ansible-playbook
inventory=None, # 关于 Inventory 实例
subset=None,
check=False, # 若为 True，只有 supports_check_mode=True 的模块才运作
```

```

diff=False,          # 是否显示 diff 模板文件的变化
environment=None,    # 环境变量 (如字典) 在命令里使用
complex_args=None,   # 结构化数据, 除了 module_args, 必须是一个字典
error_on_undefined_vars=C.DEFAULT_UNDEFINED_VAR_BEHAVIOR, # ex. False
accelerate=False,    # 是否使用加速连接
accelerate_ipv6=False, # 加速连接 w/ IPv6
accelerate_port=None, # 指定加速连接的端口
vault_pass=None,
run_hosts=None,      # 预先计算主机的可选列表上运行
no_log=False,        # 该选项用来给定任务启用 / 禁用日志记录
become=False,        # 是否可越权 become_method=C.DEFAULT_BECOME_METHOD,
become_user=C.DEFAULT_BECOME_USER,    # ex: 'root'
become_pass=C.DEFAULT_BECOME_PASS,    # ex: 'password123' 或 None
become_exe=C.DEFAULT_BECOME_EXE,      # ex: /usr/local/bin/sudo

```

由于篇幅有限, 大家可以自己做尝试。在实际使用中我们一般会选择调用 `playbook` 的方式, 在之后的内容, 会着重介绍我们最常用的 `Playbook` 的方式。

(3) Ansible 2.x 的 API 调用方法

如果读者使用的 2.0 以上版本的话, 可能会比较复杂, 但是 2.0 版本的 API 的好处也是显而易见的。对于模块非 `playbook` 的调用, 2.x 版本可以把多个模块放在一起执行, 减少连接主机所消耗的时间。

值得读者注意的是, 由于 Ansible 2.x 的程序目录和模块的整理和修改, 导致原来 1.9.x 的 API 无法使用, 使用 Ansible 2.x 的用户如果在脚本中调用 `ansible.runner` 时, 就将报该模块不存在。原因在于这个 `runner` 类已被删除, 功能被拆分到其他类中去了。

Ansible 2.x 版本我们有两种用法, 一种是使用其 API 来调用 Ansible 的模块, 另一种是使用其 API 调用 `Playbook`, 后者比较主要, 因为 `Playbook` 才是我们大规模应用的标准文件格式。在这里笔者不打算把模块封装起来, 这样代码更直接, 也方便读者们更好地学习和更改, 自行封装属于自己的 Ansible 的 API。

那么, 我们就先来介绍如何使用 2.0 的 API 来调用 Ansible 的模块。在编写上可以分为几步:

1) 导入 Ansible 2.0 中必要的模块。

```

#!/usr/bin/env python
from collections import namedtuple
from ansible.parsing.data_loader import DataLoader
from ansible.vars import VariableManager
from ansible.inventory import Inventory
from ansible.playbook.play import Play
from ansible.executor.task_queue_manager import TaskQueueManager

```

2) 定义要使用的命名空间, 传入变量等。

由于该部分是 Ansible-API 的初始化, 如果使用通用的参量, 以下代码不需要做多大变化, 读者可以略过。

```
Options = namedtuple('Options', ['listtags', 'listtasks', 'listhosts',
    'syntax', 'connection', 'module_path', 'forks', 'private_key_file', 'ssh_common_
    args', 'ssh_extra_args',
    'sftp_extra_args', 'scp_extra_args', 'become', 'become_method', 'become_user',
    'verbosity', 'check'])
variable_manager = VariableManager()
loader = DataLoader()
```

```
options = Options(listtags=False, listtasks=False, listhosts=False,
    syntax=False, connection='ssh', module_path=None, forks=100, private_key_
    file=None,
    ssh_common_args=None, ssh_extra_args=None, sftp_extra_args=None, scp_extra_
    args=None, become=False, become_method=None, become_user=None, verbosity=None,
    check=False)
passwords = {}
inventory = Inventory(loader=loader, variable_manager=variable_manager)
variable_manager.set_inventory(inventory)
```

3) 编写执行的命令。

这里把写的 echo 模块使用到其中。

```
play_source = dict(
    name = "Ansible Play",
    hosts = 'localhost', # 更改主机
    gather_facts = 'no',
    tasks = [ # 更改使用的命令
        dict(action=dict(module='shell', args='ls'), register='shell_out'),
        dict(action=dict(module='echopong', args='pong'))
    ]
)
play = Play().load(play_source, variable_manager=variable_manager,
    loader=loader)
```

4) 把上面的命令放入执行队列中执行。

```
qm = None
try:
    tqm = TaskQueueManager(
        inventory=inventory,
        variable_manager=variable_manager,
        loader=loader,
        options=options,
        passwords=passwords,
        stdout_callback='default',
    )
    result = tqm.run(play)
finally:
    if tqm is not None:
        tqm.cleanup()
```

取得我们的 API 脚本。


```
wget https://raw.githubusercontent.com/stanley1st/ansibleUI/master/Chapter_12/12-7/tag_2.1.0.0/new_api_module.py
```

关于大家比较关注的收集系统信息内容，按照上述的用法，我们也可以引出使用 `setup` 来收集机器信息的方法，通过下述脚本来收集。

请在上面 `new_api_module.py` 的步骤 1) 中添加返回日志信息的类。

```
import json
class ResultCallback(CallbackBase):
    def v2_runner_on_ok(self, result, **kwargs):
        host = result._host
        self.data = json.dumps({host.name: result._result}, indent=4)

results_callback = ResultCallback()
```

并修改步骤 3) `play_source` 部分为：

```
play_source = dict(
    name = "Ansible Play",
    hosts = 'localhost',
    gather_facts = 'no',
    tasks = [
        dict(action=dict(module='setup'))
    ]
)
```

并修改步骤 4) `tqm` 部分为：

```
tqm = TaskQueueManager(
    inventory=inventory,
    variable_manager=variable_manager,
    loader=loader,
    options=options,
    passwords=passwords,
    stdout_callback=results_callback,
)
result = tqm.run(play)
print results_callback.data
```

`results_callback.data` 的结果就是机器的所有信息的 JSON 格式，读者可以把步骤 3) 中的 `hosts = 'localhost'` 中的 `'localhost'` 修改为变量，即可满足需求。

取得我们的 API 脚本：

```
wget https://raw.githubusercontent.com/stanley1st/ansibleUI/master/Chapter_12/12-7/tag_2.1.0.0/setup_gather.py
```

上面的程序是官方提供的非调用 Playbook 的方式来调用 API，我们一般使用会比较少。接下来，我们介绍如何使用 2.0 的 API 来调用 Ansible 的 Playbook。在编写上可以分为几步。

以下我们使用到的 `test.yml` 请读者自行编写一个简单的 YML 文件即可，或使用笔者提

供的以下文件：

```
wget https://github.com/stanleyst/ansibleUI/blob/master/Chapter_12/12-7/
tag_1.9.x/test.yml
```

1) 导入 Ansible 2.0 中必要的模块，最主要的就是 PlaybookExecutor 这个模块，用于调用 Playbook 的 API。

```
# !/usr/bin/env python
import os
import sys
from collections import namedtuple
from ansible.parsing.data_loader import DataLoader
from ansible.vars import VariableManager
from ansible.inventory import Inventory
from ansible.executor.playbook_executor import PlaybookExecutor
from ansible.utils.display import log_file
import sys
```

2) 定义要使用的命名空间，传入变量等。

和之前介绍编写模块 API 时候一样，需要定义一些初始的参量到 options 中去，将下面 Options 中的全部定义上，这样可以避免由于版本的不同而导致对漏定义的报错。

```
variable_manager = VariableManager()
loader = DataLoader()

inventory = Inventory(loader=loader, variable_manager=variable_manager, host_
list='/etc/ansible/hosts')

Options = namedtuple('Options', ['listtags', 'listtasks', 'listhosts',
'syntax', 'connection', 'module_path', 'forks', 'private_key_file', 'ssh_
common_args', 'ssh_extra_args', 'sftp_extra_args', 'scp_extra_args', 'become',
'become_method', 'become_user', 'verbosity', 'check'])
options = Options(listtags=False, listtasks=False, listhosts=False,
syntax=False, connection='ssh', module_path=None, forks=100, private_key_
file=None, ssh_common_args=None, ssh_extra_args=None, sftp_extra_args=None,
scp_extra_args=None, become=False, become_method=None, become_user=None,
verbosity=None, check=False)
```

3) 为 API 注入 Playbook 的路径和参数。

```
playbook_path = 'test.yml'
variable_manager.extra_vars = {"aa": "xx", "bb": [{"xx": "1"}]}
if not os.path.exists(playbook_path):
    print '[INFO] The playbook does not exist'
    sys.exit()
```

4) 执行并打印结果。

```
passwords = {}
```

```

pbex = PlaybookExecutor(playbooks=[playbook_path], inventory=inventory,
variable_manager=variable_manager, loader=loader, options=options,
passwords=passwords)
code = pbex.run()
stats = pbex._tqm._stats
hosts = sorted(stats.processed.keys())
result = [{h: stats.summarize(h)} for h in hosts]
results = {'code': code, 'result': result, 'playbook': playbook_path}
print(results)

```

取得我们的 API 脚本。

```

wget https://raw.githubusercontent.com/stanley1st/ansibleUI/master/
Chapter_12/12-7/tag_2.1.0.0/new_api_playbook.py

```

上面的模块虽然能够返回结果，但却不能返回我们的 Ansible 的执行日志，这样，必将使得当 Ansible 调用出现问题后，我们很难快速定位 YML 文件的 BUG 所在。这样的调用 API 写的脚本或程序得到不完整的结果反馈，也必将得不到最终用户认可。

在这里我们就需要修改源码了，所需修改的地方不多，首先我们需要找到我们使用的 Ansible 模块的目录，还是分 1.9.x 和 2.x 来介绍两种情况；另外还会分为仅取得日志文件内容和实时写入文件进行保存和记录两种情况。

1.9.x 的取得日志文件内容的源码修改的实现如下：找到你所使用的 Ansible 模块，如果使用虚拟环境，请到你的虚拟环境中找到 Ansible 进行修改。比如笔者使用的是虚拟环境，虚拟环境目录在 /data/env/ 下，那么笔者所改的文件应该是 /data/env/lib/python2.7/site-packages/ansible/callbacks.py。

```

callback_plugin.task = task          # 搜索这一行，空开一行，在其后面加入 log_add = []
log_add = []
def display(msg, color=None, .....)

```

另一处需要修改的地方如下：

```

log_unflock(runner)                  # 搜索这一行，在其后面添加以下内容
try:
    log_add.append(msg2)
except:
    pass

```

```

def call_callback_module(method_name, *args, **kwargs):

```

完成了以上两步修改后，就可以开始使用我们的 API 来生成日志了。笔者已经封装好了 Playbook 的 API，读者直接使用 Ansi_Play 这个模块就可以了。

```

excute = Ansi_Play('test.yml',{'test': '/tmp/test.book'})
# Ansi_Play('YML 文件名', 为 Ansible 的 YML 的 {} 传递字典参数)
result = excute.run()                # result 为详细的信息

```

Ansi_Play 模块展示如下:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
import ansible.runner
import ansible.playbook
import ansible.inventory
from ansible import callbacks
from ansible import utils
import re

class Ansi_Play(object):

    def __init__(self, playbook, extra_vars={}): # 初始化参数
        self.stats = callbacks.AggregateStats()
        self.playbook_cb = callbacks.PlaybookCallbacks(verbose=utils.VERBOSITY)
        self.extra_vars = extra_vars
        self.playbook = playbook
        self.setbook = self.book_set()

    def book_set(self): # 使用 playbook 模块
        runner_cb = callbacks.PlaybookRunnerCallbacks(self.stats, verbose=utils.VERBOSITY)
        self.pb = ansible.playbook.PlayBook(
            playbook = self.playbook,
            stats = self.stats,
            extra_vars = self.extra_vars,
            callbacks = self.playbook_cb,
            runner_callbacks = runner_cb
        )

    def ansi_escape(self, text): # 移除 Linux 字体颜色
        ansi_escape = re.compile(r'\x1b[^\m]*m')
        return ansi_escape.sub('', text)

    def run(self):
        simple = self.pb.run()
        detail = self.ansi_escape('\n'.join(callbacks.log_add))
        # 添加日志内容到 detail
        return {'simple': simple, 'detail': detail}

if __name__ == "__main__":
    excute = Ansi_Play('test.yml',{'test': '/tmp/test.book'})
    result = excute.run()
    print result
```


运行后打出来的实时日志程序是取不到的, 只有最后 result 这个参数 print 出来的 Ansible 日志内容, 我们的程序才能取到 (即 result)。

取得我们的 API 脚本。


```
wget https://raw.githubusercontent.com/stanley1st/ansibleUI/master/Chapter_12/12-7/tag_1.9.x/Ansi_play.py
```

通过执行 `python Ansi_play.py`, 我们所得出的结果如下:

```
{'simple': {'localhost': {'unreachable': 0, 'skipped': 0, 'ok': 2, 'changed': 0,
'failures': 0}}, 'detail': u'\nPLAY [test] *****
***** \n\nTASK: [echo pongpong] *****
***** \nok: [localhost]\n\nTASK: [debug var=res] *****
***** \nok: [localhost] => {\n      "var":
\n      "res": {\n          "changed": false, \n          "echo": "pong",
\n          "invocation": {\n              "module_args": "args=\\\"pong\\\"", \n
"module_complex_args": {}, \n              "module_name": "echo"\n          }\n
\n      }\n}'}
```

 **注意** result 结果 {'simple': 简要结果, 'detail': 执行日志 }。

对 1.9.x 的实时写入文件进行保存和记录的源码进行修改的原因是: 我们在写程序的时候总是希望能够保存日志到文件中, 但同时我们还需要解决一个问题, 单单使用上面的方法, 就只能等到 YML 执行完后才返回该 YML 的执行日志, 这对于要执行很长时间的 YML 文件来说, 就将产生一个不知道中间运行结果的真空期。如果该日志文件能够保证被实时写入, 那就能够将把我们上面取到的日志文件内容不断地反馈给用户。接下来我们就来解决这个问题。

笔者的 Ansible 的 callbacks 模块的修改, 读者自行修改自己使用的 Ansible 目录下的 callbacks 模块 `/data/env/lib/python2.7/site-packages/ansible/callbacks.py`。

```
callback_plugin.task = task # 搜索这一行, 空开一行, 在其后面加入 log_add = []
```

```
import re
log_add = []
log_file = []
```

另一处需要修改的地方如下:

```
log_unflock(runner) # 搜索这一行, 在其后面添加
try:
log_add.append(msg2)
except:
    pass
if log_file:
    if msg2.find('before assignment') == -1: # 排除变量未提交警告
        with open(log_file[0], 'a+') as f:
            ansi_escape = re.compile(r'\x1b[^m]*m') # 去除 Linux 颜色标识
            f.write(ansi_escape.sub('', msg2))
```

对于调用方法可以查看 `Ansi_play_log.py`。

```
def run(self, log_file=''): # 增加实时文件记录
    if log_file:
        callbacks.log_file.append(log_file)
    simple = self.pb.run()
    detail = self.ansi_escape('\n'.join(callbacks.log_add))
    # 添加日志内容到 detail
    return {'simple': simple, 'detail': detail}

if __name__ == "__main__":
    excute = Ansi_Play('test.yml',{'args': 'pong'})
    result = excute.run('/tmp/test.log')
    print result
```

取得我们的 API 脚本:

```
wget https://raw.githubusercontent.com/stanley1st/ansibleUI/master/Chapter_12/12-7/tag_1.9.x/Ansi_play_log.py
```

通过执行 `python Ansi_play_log.py`, 我们除了得到上面相同的信息外, 还会记录日志到 `/tmp/test.log` 中去, 而且是实时日志。笔者这里的例子执行时间比较短, 读者可以写一个执行时间较长的 YML, 执行后使用 `tail -f /tmp/test.log` 来查看结果。

对于 2.x 的仅取得日志文件内容的源码的修改如下: 基本思想和 1.9.x 的修改思路是一样的, 请读者自行修改自己使用的 Ansible 目录下的 `callbacks` 模块 `/data/env/lib/python2.7/site-packages/ansible/ utils/display.py`。

```
import re
log_add = []
log_file = []
class Display: # 找到该行, 在上面添加 3 行, 注意空行
```

另外还需要在该文件中添加以下内容。

```
logger.info(msg2) # 找到该行, 在下面添加代码
if msg2.find('before assignment') == -1:
    ansi_escape = re.compile(r'\x1b[^\m]*m')
    msg3 = ansi_escape.sub('', msg2)
else:
    msg3 = ''
try:
    log_add.append(msg3)
except:
    pass
if log_file:
    with open(log_file[0], 'a+') as f:
        f.write(msg3)
def v(self, msg, host=None): # 找到此行, 在上面添加代码, 请添加在 display 里面
    return self.verbose(msg, host=host, caplevel=0)
```

现在我们来调用 Ansible2.x 的模块。这里请读者注意, 由于 Ansible2.x 更新 API 的速

度比较快，如果读者使用的版本比较高，请以本书 GitHub 上第 12 章的更新的 API 为准。笔者使用的 Ansible 版本为 2.1.0.0，由于笔者已经对其封装过了 `Ansi_play2.py`，内容比较多，我们还是分批进行介绍。如果读者觉得以下内容比较难懂，可以直接调用即可，无须看懂。调用方式请参看 GitHub，或者查看该部分最后的 `if __name__ == '__main__':` 后的内容。

步骤 1：导入 Ansible 所需的 API 和 Python 的一些基础模块。

```
# !/usr/bin/env python

import os
import sys
import re
import json
from collections import namedtuple

from ansible.parsing.data_loader import DataLoader
from ansible.vars import VariableManager
from ansible.inventory import Inventory
from ansible.executor.playbook_executor import PlaybookExecutor
from ansible.utils.display import log_file, log_add
from ansible.plugins.callback import CallbackBase
from ansible.errors import AnsibleParserError
```

步骤 2：使用 Ansible 的 callbacks 插件，按机器的执行情况（成功、失败或者未连接）分类返回如下日志。

```
class ResultCallback(CallbackBase):
    ''' if needed, you can modify it yourself'''
    def __init__(self, *args):
        super(ResultCallback, self).__init__(*args)
        self.ok = json.dumps({})
        self.fail = json.dumps({})
        self.unreachable = json.dumps({})
        self.playbook = ''
        self.no_hosts = False

    def v2_runner_on_ok(self, result): # 执行成功的 playbook 日志
        host = result._host.get_name()
        self.runner_on_ok(host, result._result)
        data = json.dumps({host: result._result}, indent=4)
        self.ok = data

    def v2_runner_on_failed(self, result, ignore_errors=False):
        # 执行失败的 playbook 日志
        host = result._host.get_name()
        self.runner_on_failed(host, result._result, ignore_errors)
        data = json.dumps({host: result._result}, indent=4)
        self.fail = data
```



```
    verbosity=None, check=check)
```

步骤 4: 不使用 Ansible 插件，返回日志简单结果，并记录日志到文件的 run 方法。

```
# 注：仍然在 class Ansi_Play2 中
def run(self, log):
    if log:
        # 设定使用哪个文件来记录日志
        log_file.append(log)
    if not os.path.exists(self.playbook):
        code = 1000
        # playbook 的 yml 文件不存在则返回 code 1000
        results = 'not exists playbook: ' + self.playbook
        return code, results, None
    pbex = PlaybookExecutor(playbooks=[self.playbook],
                             inventory=self.inventory,
                             variable_manager=self.variable_manager,
                             loader=self.loader,
                             options=self.options,
                             passwords=self.passwords)
    try:
        code = pbex.run()
    except AnsibleParserError:
        code = 1001
        # playbook 的 yml 文件语法错误则返回 1001
        result = 'syntax problems in ' + self.playbook
        return code, results, None
    stats = pbex._tqm._stats
    hosts = sorted(stats.processed.keys())
    results = [{h: stats.summarize(h)} for h in hosts]
    if not results:
        code = 1002
        # 没有匹配到任何主机名则返回 1002
        result = 'no host executed in ' + self.playbook
        return code, results, None
    complex = '\n'.join(log_add)
    # 取得完整日志信息
    return code, results, complex
```

步骤 5: 使用插件, 可以返回分类日志, 不记录日志文件的 run_need_data 方法。

```
def run_need_data(self):
    # 与上面大致相同，但加了插件功能
    if not os.path.exists(self.playbook):
        code = 1000
        complex = {'playbook': self.playbook,
                    'msg': self.playbook + ' playbook does not exist', 'flag': False}
        simple = 'playbook does not exist about ' + self.playbook
        return code, simple, complex
    pbex = PlaybookExecutor(playbooks=[self.playbook],
                             inventory=self.inventory,
```

```

        variable_manager=self.variable_manager,
        loader=self.loader,
        options=self.options,
        passwords=self.passwords)
    results_callback = ResultCallback() # 使用 ResultCallback 插件
    pbex._tqm._stdout_callback = results_callback
    try:
        code = pbex.run()
    except AnsibleParserError:
        code = 1001
        complex = {'playbook': self.playbook,
                   'msg': 'syntax problems in ' + self.playbook, 'flag': False}
        simple = 'syntax problems in ' + self.playbook
        return code, simple, complex
    if results_callback.no_hosts:
        code = 1002
        complex = 'no hosts matched in ' + self.playbook
        simple = {'executed': False, 'flag': False, 'playbook': self.playbook,
                  'msg': 'no_hosts'}
        return code, simple, complex
    else:
        ok = json.loads(results_callback.ok)
        fail = json.loads(results_callback.fail)
        unreachable = json.loads(results_callback.unreachable)
        if code != 0:
            complex = {'playbook': results_callback.playbook, 'ok': ok,
                       'fail': fail, 'unreachable': unreachable, 'flag': False}
            simple = {'executed': True, 'flag': False, 'playbook': self.playbook,
                      'msg': {'playbook': self.playbook, 'ok_hosts': ok.keys(), 'fail':
                              fail.keys(), 'unreachable': unreachable.keys()}}
            return code, simple, complex
        else:
            complex = {'playbook': results_callback.playbook, 'ok': ok,
                       'fail': fail, 'unreachable': unreachable, 'flag': True}
            simple = {'executed': True, 'flag': True, 'playbook': self.playbook,
                      'msg': {'playbook': self.playbook, 'ok_hosts': ok.keys(),
                              'fail': fail.keys(), 'unreachable': unreachable.keys()}}
            return code, simple, complex

```

步骤 6：调用上述 `run` 和 `run_need_data` 两个方法，我们所有方法的返回值都是 3 个：`code`、`simple`、`complex`（返回状态码、简单返回结果、所有日志信息）。通过执行下面的程序可以得到所有返回值和信息：

```

if __name__ == '__main__':
    book2 = Ansi_Play2('test.yml')
    # 用户可以选择下面任意一种方法
    # 如果你要取得数据，并把这些数据与自己的程序做结合分析，请选择下面这种插件方式
    code, simple, complex = book2.run_need_data()
    print code, simple, complex

```

```

# 返回值 code, simple, complexzh
# code 0
# simple: {'msg': {'fail': [], 'unreachable': [], 'ok_hosts': [u'localhost'],
'playbook': 'test.yml'}, 'flag': True, 'executed': True, 'playbook':
'test.yml'}
# complex: {'code': 0, 'ok': {u'localhost': {u'invocation': {u'module_name': u'debug',
u'module_args': {u'var': u'res'}}}, u'res': {u'changed': True, u'end': u'2016-07-
16 00:29:42.936217', u'stdout': u'123', u'cmd': u'bash -c "echo 123; sleep
10"', u'rc': 0, u'start': u'2016-07-16 00:29:32.932963', u'stderr': u'', u'delta':
u'0:00:10.003254', u'stdout_lines': [u'123'], u'warnings': [], u'changed': False,
u'_ansible_verbosity': True, u'_ansible_no_log': False}}, 'flag': True,
'playbook': u'test', 'fail': {}, 'unreachable': {}}
# 如果你只需要知道脚本是否正确执行, 并需要执行并收集实时日志到我们定义的 /tmp/aa.log 文件中,
下面这种更合适你
code, simple, complex = book2.run('/tmp/aa.log')
print code, simple, complex

# 返回值 code, simple, complexzh
# code 0
# simple: [{u'localhost': {'unreachable': 0, 'skipped': 0, 'ok': 2, 'changed':
1, 'failures': 0}}]
# complex 即 log_file: '/tmp/aa.log' 中内容

```

取得我们的 API 脚本:

```
wget https://raw.githubusercontent.com/stanley1st/ansibleUI/master/Chapter_12/12-7/tag_2.1.0.0/Ansi_play2.py
```

通过把 Ansible 的模块和 API 的互相配合使用, 我们已经可以写一些相对复杂的应用了, 这与传统 Shell 命令相比, 我们可以不去考虑它的多线程的编写了。与单纯调用 Ansible 的基础模块不同, 我们可以使用它的返回值了, 这对于后续大规模程序编写比较重要。

12.8 本章小结

通过本章节的学习, 我们学会了如何编写 Ansible 模块, 并把它应用在命令行及 playbook 中。之后我们又学习了 Ansible 的 API 的调用, 为后续的程序应用打下了基础。这两者对于想要提升自己能力的 Ansible 用户来说, 缺一不可。

对于模块编写, Ansible 提供了很好的模块编写方法, 除了一些文件传输和同步的模块以外, 大多数模块都是按照 1 ~ 6 节中的内容进行编写的。因此, 通过学习本章内容, 或者通过参照 Ansible 已有的模块进行模仿, 可以编写更适合自己应用的模块, 这不仅仅可以加强自己自身的脚本编写能力, 也可以为团队乃至 Ansible 官方提供自己的模块, 为社区贡献自己的一份力量。

对于 Ansible 的 API 调用, 我们介绍了 1.9.x 和 2.1.x 两个版本的使用法, 还指出了这两者之间的差异和用法上的区别。对于 API 调用上, 推荐使用 API 调用 Playbook, 原因在于 Playbook 学习成本低, 任何接触 Ansible 的用户查阅资料后都可自行编写, 我们只要提供一个接口去调用 Playbook 即可, 这样就可以完成工作上的分工, 加快各自的工作进度。

该章节的内容为大规模构建集成程序做了很有用的基础工作, 为完成我们后续的自动化运维的编写任务铺平了道路。

开发自己的 Ansible WebUI

我们在之前学习到了如何使用 Ansible，但也只是局限于使用命令和脚本执行 Ansible 的命令，这对于不断复杂的运维体系来说是远远不够的。现在使用 Web 可视化来完成操作也是当今乃至未来很长一段时间的方向。所以本章我们将主要介绍将 Ansible 与 Web 界面相结合的使用方法。

13.1 搭建 Django 开发环境

既然要使用 Web 应用，我们就要在很多 Web 框架中选型，比如 Web.py、Flask、Bottle、Django 等，在所有 Web 框架中，笔者还是比较推荐 Django 的，因为它是一款开箱即用的软件，无需太多的插件便可完成基本上所有 Web 开发，比较适合刚刚入门 Web 应用开发的人员。

13.1.1 为什么要使用 Web 页面做管理

由于 Ansible 的使用需要我们对 Linux 有所了解，另外，和所有 Shell 脚本一样，我们所写的一些 Playbook 的 YML 文件更需要一个熟悉它的人进行操作，这就间接造成我们一些操作的危险性和同事之间沟通而导致的效率问题。对于这些问题都会在无形中增加我们的运维成本。既然是这样，应该怎么去解决这个问题呢？对于这个问题，我们有很多方法来解决，在这里，笔者认为需要构建我们自己的 Web 页面，做成可视化的应用，这样就可以轻松完成我们日常的一些简单操作了。

对于可视化，我们有以下几种选择。

1) 选择使用 Jenkins 来完成。Jenkins 已经为我们提供了页面，只要把调用 Ansible 命令的 Shell 脚本放入其中就可以了。这个对于非开发型或中小规模的企业，已经绰绰有余了，但对于需要构建复杂控制流来进一步减少人为事故的运维团体来说，Jenkins 是不够的。所以我们不会对 Jenkins 的使用做介绍，网上的资料比较多，笔者就不详述了。

2) 可以购买 Ansible 的 Tower 产品，如果针对想要构建契合自己企业应用的应用平台的话，就可以选择，编写自己的页面了。这个可能是一个比较漫长的过程，需要反复修改和改版，但优势也很明显，完全是符合自己企业需求开发的，可用性会比较高。

3) 构建适合自己企业的运维应用，那么需要使用 Ansible 来构建 Web 应用。对于为什么选用 BS 架构 (Browser-Server)，而不使用 CS (Client-Server) 架构，是因为我们可以直接使用 Browser 作为客户端，不必强迫用户安装我们自己编写的 Client 了。那么前提就是，我们应该学习一个 Web 框架，在这里我们选用了 Django 作为我们后面提供 Web 接口的一个应用。之所以使用 Django，是因为它已经为大家构建好了完整的开发环境，不需要我们找过多的第三方的支持库，对于初学者来说可以迅速上手。接下来，就开始准备我们的环境吧！

13.1.2 系统及软件环境

1) 选用 OS 系统和 Python 环境。在这里我们挑选 Ubuntu 14.04，读者可以挑选自己喜欢的 OS 系统，但对于一些系统软件的安装，各个 OS 系统可能软件名称不一样，需要大家网上查找下。另外，我们还是以大家用得最多的 Python 2.7.x 来做开发环境。

2) 安装的基础软件：MySQL、Git、Virtualenv、pip (这里我们就不用源码安装了)。

```
sudo apt-get install mysql-server git python-virtualenv python-pip ansible -y
```

3) 安装的相关开发环境依赖包。

```
sudo apt-get install python-dev libmysqlclient-dev -y
```

4) 创建虚拟环境，并安装使用 Python 模块。创建 Python 开发的虚拟环境，可以有效地隔离对系统 Python 依赖的影响，换句话说，就算我们把虚拟环境的 Python 依赖弄坏，机器也不会受到影响。

```
virtualenv demo
source demo/bin/activate # 进入虚拟环境
pip install ansible==1.9.4 Django==1.8 django-filter djangorestframework==3.2.3
MySQL-python# 在进入虚拟环境后再安装这些 Python 模块
```

如果以上有遇到版本不存在的问题，并更改版本号 (比如：ansible==1.9.2)，安装的软件大致有以下这些就可以了。

```
(demo)demouser@demouser-virtual-machine:~$ pip list
```

```

ansible (1.9.4)
argparse (1.2.1)
Django (1.8)
django-filter (0.13.0)
djangorestframework (3.2.3)
Jinja2 (2.8)
MySQL-python (1.2.5)
paramiko (2.0.0)
pip (1.5.4)
pyasn1 (0.1.9)
pycrypto (2.6.1)
PyYAML (3.11)
setuptools (2.2)
wsgiref (0.1.2)

```

5) 使用 Git 得到我们的基础 Django 框架的 demo。

```
git clone https://github.com/targetoyes/book-demo.git demo-app
```

13.2 Django 配置文件详解

对于 Django，笔者不准备做大量的介绍，只介绍我们用到的一些应用。如果觉得不够用可以自行查阅 Django 的官方网站进行补缺。另外，如果不想被后端的编写规则所困扰的话，推荐大家关注一个 Django 的第三方库，这也是国外用得比较多的一个框架：Restframework。

13.2.1 Django 的基础配置及运行

大家在 13.1.2 节已经把我们 Git 上的 demo 文件给下载下来了，接下来将围绕这个 demo 对 Django 的这个配置进行介绍。

先将我们的 Django 框架运行起来。

- 1) 正确连接数据库，并导入 demo 的数据库结构。
- 2) 修改数据库连接的配置文件 demo-app/ansible_demo/root_demo/local_settings.py。

```

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql', # Add 'postgresql_psycopg2', 'postgresql',
        'mysql', 'sqlite3' or 'oracle'.
        'NAME': 'demo', # Or path to database file if using sqlite3.
        'USER': 'root', # Not used with sqlite3.
        'PASSWORD': 'root1234', # Not used with sqlite3.
        'HOST': '', # Set to empty string for localhost. Not used with sqlite3.
        'PORT': '3306', # Set to empty string for default. Not used with sqlite3.
    }
}

```

- 3) 在 MySQL 数据库中创建一个数据库名（在这里我们使用的是 demo，请和上面数据

库配置的 'NAME':'demo' 保持一致)。

4) 确保你仍然在虚拟环境中, 并使用 Django 的数据结构导入功能。

```
cd demo-app/ansible_demo/
./manage.py migrate
```

5) 尝试启动 Django 应用, 如果没有报错那就代表启动成功。我们推荐使用 screen 来启动 Django 的应用, 相当于使用后台执行, 但我们可以实时看到它的日志(如无 screen 请自行安装)。

```
screen -S demo
source demo/bin/activate
cd demo-app/ansible_demo/
./manage.py runserver 0.0.0.0:8001
```

如果有以下提示, 那么初始工作全部做完了。

```
Performing system checks...

System check identified no issues (0 silenced).
May 08, 2016 - 22:29:40
Django version 1.8, using settings 'root_demo.settings'
Starting development server at http://0.0.0.0:8001/
Quit the server with CONTROL-C.
```

13.2.2 Django 的主配置目录介绍

我们在 13.2.1 节已经把 Django 应用成功地运行起来了。接下来, 我们简单地介绍 Django 主配置目录 (demo-app/ansible_demo/root_demo) 的一些目录及配置文件。

我们把 Django 的原来的配置文件拆成了 4 个文件及路由配置文件。

(1) dev_settings.py 文件

我们在这里设置了 2 个重要的配置:

❑ DEV_INSTALLED_APPS 配置要注入的 app (注入了 demo_1)。

❑ REST_FRAMEWORK 配置序列化的 Django 插件, 它将提高我们的编程速度, 让我们专注于编写逻辑。

(2) local_settings.py 文件

这个配置文件将储存我们所有的前端路径和固定连接方式。

❑ STATIC_ROOT 定义静态文件路径。

❑ TEMPLATE_ROOT 定义 html 模板路径。

❑ MEDIA_ROOT 定义下载路径。

❑ DATABASES 配置我们和数据库的连接。

❑ CACHES 与 Redis 或 memcache 的连接。

□ LOGGING 为日志的输出方式。

(3) prod_settings.py 文件

正式环境切换配置（我们暂时用不到）。

(4) settings.py 文件

定义所有 Django 的所有插件的路径及时区。

(5) urls.py 文件

定义我们的根路由 static、media，以及我们将自己编写的 app (demo_1)。

13.2.3 Django 的 app 目录介绍

接下来，我们介绍一下前端配置的目录。

(1) demo-app/ansible_demo/root_demo /static_root 目录

为了便于管理，我们将它分成 3 个目录进行归类：CSS 文件目录、JavaScript 文件目录、fonts 字体文件目录。以后用到的插件都可以往下面放。

(2) demo-app/ansible_demo/root_demo /templates 目录

我们主要存放 html 的文件目录，通过导入上面各个页面所要的静态文件，来实现动态页面，另外我们只使用 Django 的页面层的标签（如 {% block %}，{% include %}），其他的逻辑标签（如 {% if %} 等），由于没有 JS 的逻辑效果好，一律不使用。

(3) demo-app/ansible_demo/demo_1 目录

这是一个子 app 目录，需要将这个 app 目录加到 dev_settings.py 里面的 DEV_INSTALLED_APPS 中才能生效。

□ demo-app/ansible_demo/demo_1/urls.py 文件：定义 app 使用的 url 路由。

□ demo-app/ansible_demo/demo_1/views.py 文件：定义 app 的视图，起到连接 html 与 url 关联的纽带作用。

□ demo-app/ansible_demo/demo_1/api/models.py 文件：定义数据库的表结构后，使用 makemigrations 和 migrate 来快速生成数据库表结构（1.7 版本以下请使用 south 来生成表结构）。

□ demo-app/ansible_demo/demo_1/api/serializers.py 文件：把数据库的数据序列化，便于我们获取，在安装 Restframework 这个框架后才可以使使用。

□ demo-app/ansible_demo/demo_1/api/api_demo_1.py 文件：后端的一切接口逻辑都可以在此处编写。

(4) demo-app/ansible_file 目录

为了对 Ansible 的 Playbook 做统一管理，我们把所有 Playbook 的 YAML 文件放在该目录，与 Django 框架无关，只是为我们自己定义一个统一目录。

13.3 编写 Ansible 的 Web 接口

在编写页面之前，我们需要学会编写 Web 接口来给前端 JavaScript 使用，我们主要编写的文件的目录在 `demo-app/ansible_demo/demo_1/api/ api_demo_1.py`，这个我们在 13.2 节的末尾已经介绍过了。为了更清晰地为大家介绍，笔者准备把它细化成 8 个问题来解决。

- 1) 如何调用接口在服务器本地创建一个文件 (touch 函数)?
- 2) 如何调用接口并使用 Ansible 命令在远程创建一个文件 (touch_2 函数)?
- 3) 如何检查接口命令被正确调用 (touch_3 函数)?
- 4) 如何记录 Ansible 命令的实时日志 (touch_4 函数)?
- 5) 如何避免调用 Ansible 命令记录实时日志时，导致检查接口命令的方法失效 (touch_5 函数)?
- 6) 如何给接口传入我们的参数 (touch_6 函数)?
- 7) 如何调用我们的 Playbook 文件的接口 (touch_7 函数)?
- 8) 在给 Playbook 传入参数时，如何解决导致检查接口命令的方法失效的问题 (touch_8 函数)?

好，那么我们就开始一步一步解决我们的上述的问题吧!

第 1 个问题比较简单。我们来假定一个场景：

我们来编写 Web 接口。首先在本地服务器上的 `tmp` 目录下创建一个文件 `abc`，这时候我们使用 `Restframework` 这个 `Django` 的扩展框架来完成自动路由的功能（自动创建 `url` 无须再到 `urls.py` 里进行定义），在 `DemoViewSet` 这个类里，使用 `restframework` 框架的装饰器后便可以编写我们的接口文件。

```
@list_route(methods=['get', 'post'])
def touch(self, request):
    os.system('touch /tmp/abc')
    return Response({'msg': '/tmp/abc has been touched'})
```

我们在浏览器上访问 `ansible_demo/ demo_1/api/api_demo_1.py` 中的 `touch` 方法，调用时请将下面的 `your_ip_address` 更改成读者自己的服务器地址。

```
http://your_ip_address:8001/demo_1/demo_api/touch/
```

然后，我们再去查看 `tmp` 目录下是否有 `abc` 这个文件。这样，我们就简单地写出了一个 `http` 方式的可供调用的 API 了。

那么第 2 问题，根据这个思路，就可以把我们的 `ansible` 命令放入其中了。

```
@list_route(methods=['get', 'post'])
def touch_2(self, request):
    os.system('ansible localhost -a "touch /tmp/bcf"')
    return Response({'msg': '/tmp/bcf has been touched'})
```

同样，我们在浏览器上访问 `ansible_demo/ demo_1/api/api_demo_1.py` 中的 `touch_2` 方法。

`http://your_ip_address:8001/demo_1/demo_api/touch_2/`

在浏览器上输入后，我们在服务器上查看 `tmp` 目录下是否有 `bcf` 这个文件。如果存在，则证明上面的 API 是正确的。

这样，我们就掌握了最简单的 Django 与 Ansible 结合的使用方法了。

但有一个问题，我们没法知道命令是否被正确执行，我们该如何判断呢？

这就是我们的第 3 个问题。所幸 Python 有一个模块给我们提供了便利，那就是 `commands` 模块^①，可以使用这个模块来判断 Ansible 命令或者 Playbook 是否被执行，而且有没有发生错误。来验证一下，我们在一个不存在的目录下面创建一个文件，它应该会出现错误。

```
@list_route(methods=['get', 'post'])
def touch_3(self, request):
    bits = 'ansible localhost -a "touch /tmp/eee"'
    (status, output) = commands.getstatusoutput(bits)
    if status != 0:
        return Response({'msg': '/tmp/eee has not been touched', 'output': output})
    return Response({'msg': '/tmp/eee has been touched', 'output': output})
```

那我们来再试验一下，还是来访问我们的 url。

`http://your_ip_address:8001/demo_1/demo_api/touch_3/`

这时候 Django 的 debug 页面会显示返回信息，如果返回是 `not been touched`，那么我们的试验就成功了。

```
HTTP 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, POST, HEAD, OPTIONS
{
  "msg": "/tmp/eee has not been touched",
  "output": "127.0.0.1 | FAILED | rc=1 >>\ntouch: 无法创建 \"/tmp/eee\"：没有那个文件或目录\n"
}
```

通过这种简单的方法，我们就有效地规避了错误。

如果我们要记录这些日志怎么办？即第 4 个问题。那太容易了，Linux 里有一个 `tee` 命令，我们可以使用它。

```
@list_route(methods=['get', 'post'])
def touch_4(self, request):
    bits = 'ansible localhost -a "touch /tmp/eee"|tee -a /tmp/ansible.log'
    (status, output) = commands.getstatusoutput(bits)
    if status != 0:
```

① 在 Python 3 中该模块被删除了，但我们可以找到与之相同的模块 `subprocess`，可以使用 `subprocess.getstatusoutput` 来替换后面用到的 `commands.getstatusoutput`，功能大致一样。

```

return Response({'msg': '/tmp/eee has not been touched', 'output': output})
return Response({'msg': '/tmp/eee has been touched', 'output': output})

```

我们可以去看看 `/tmp/ansible.log` 这个日志文件是否存在。

`http://your_ip_address:8001/demo_1/demo_api/touch_4/`

通过这种方法，我们就有效而且实时地保存了日志文件，之后我们可以再去写一个接口，每隔几秒去读取这个日志文件，这样我们就可以取得实时的日志了。下面是该 API 最后返回的结果：

```

HTTP 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, POST, HEAD, OPTIONS

{
  "msg": "/tmp/eee has been touched",
  "output": "127.0.0.1 | FAILED | rc=1 >>\ntouch: 无法创建 \"/tmp/eee\": 没有那个文件或目录\n"
}

```

但是，看看结果，大家是不是发现有什么不对？应该返回的是 `"msg": "/tmp/eee has not been touched"` 才对，这是怎么回事呢？因为如果使用 `tee` 的话，所有的命令执行的状态码都将是 0。那我们有什么好的方法解决吗？接下来我们就来解决第 5 个问题。我们可以在每条命令前面加上 `set -o pipefail`，就可以了。

```

@list_route(methods=['get', 'post'])
def touch_5(self, request):
    bits = 'set -o pipefail;ansible localhost -a "touch /tmp/eee"|tee -a /tmp/ansible.log'
    (status, output) = commands.getstatusoutput('bash -c "{0}"'.format(bits))
    if status != 0:
        return Response({'msg': '/tmp/eee has not been touched', 'output': output})
    return Response({'msg': '/tmp/eee has been touched', 'output': output})

```

访问一下我们刚刚编写的 url 接口：

`http://your_ip_address:8001/demo_1/demo_api/touch_5/`

那么我们来执行修改之后的文件，已经能得到了我们想要的正确结果了。

```

HTTP 200 OK
Content-Type: application/json
Vary: Accept
Allow: GET, POST, HEAD, OPTIONS

```

```

{
  "msg": "/tmp/eee has not been touched",

```



```
"output": "127.0.0.1 | FAILED | rc=1 >>\ntouch: 缺少了文件操作数\nTry 'touch
--help' for more information.\n"
}
```

以上我们都是直接访问网页接口的，相当于直接使用了该接口的 `get` 方法。那么我们怎么使用它的 `post` 方法来传入我们自己定义的一些变量呢？

我们来解决第 6 个问题。我们会用到 `restframework` 中的 `request` 这个参数，所有的 `post` 参数都存放在 `request.data` 之中，这样我们可以很简单地进行调用。

我们现在想要自己定义创建的文件，就要把我们现在的内容修改成 `touch_6` 函数定义的方式，但我们不能用 `http://your_ip_address:8001/demo_1/demo_api/touch_6/` 来调用接口，这样无法传入自己的参数。我们可以使用 `curl` 来完成。我们的接口如下：

```
@list_route(methods=['get', 'post'])
def touch_6(self, request):
    data = request.data
    filename = data['filename']
    bits = 'set -o pipefail;ansible localhost -a "touch {0}"|tee -a /tmp/ansible.
log'.format(filename)
    (status, output) = commands.getstatusoutput('bash -c "{0}"'.format(bits))
    if status != 0:
        return Response({'msg': '{0} has not been touched'.format(filename), 'output':
output})
    return Response({'msg': '{0} has been touched'.format(filename), 'output': output})
```

我们使用 `curl` 来 `post` 我们的数据。

```
curl -X post http://your_ip_address:8001/demo_1/demo_api/touch_6/ -H "Content-
Type: application/json" -d '{"filename": "/tmp/sda"}'
```

我们得到的结果如下：

```
{"msg": "/tmp/sda has not been touched", "output": "127.0.0.1 | FAILED | rc=1
>>\ntouch: 缺少了文件操作数\nTry 'touch --help' for more information.\n"}
```

在这里我们不使用 `Ansible` 的 `API` 的方式来调用 `Ansible` 命令，之所以不使用 `Ansible` 的 `API` 的方式进行编写，是因为这样我们同样完成了我们的功能，使用了最简单易懂的方式实时地得到我们的日志文件了，同时也照顾了大多数读者。如果大家想要使用 `API` 的方式，那么请读者细读地第 12 章的内容，把上面的代码进行更改，调用第 12 章最后一节封装后的方法（请注意版本 1.9.x 或 2.1.0）。再则如果有一个耗时很长的命令，我们总是希望能够得到它执行的实时信息，这样，我们可以另写一个方法去实时地读取这个日志文件。如果读者学会第 12 章前面几节关于 `Ansible` 模块的编写，就可以在 `Playbook` 中使用我们自己编写的模块了。另外需要提醒的是，对于喜欢使用异步 `gevent` 来启动 `Django` 应用的读者来说，当调用到这个多线程 `Ansible` 的 `API` 时一定会报错，所以若你是此类用户的话，请使用 `Shell` 的方式来调用 `Ansible`。

我们前面的铺垫都是为了最后一步，来解答我们的第 7 个问题。在这之前，先给我们的 Playbook 编写 Web 接口。我们先准备好以下 Playbook：

```
- hosts: localhost
```

```
tasks:
```

```
- name: touch file
```

```
shell: touch /tmp_1/ggg
```

我们来看以下接口函数：

```
@list_route(methods=['get', 'post'])
```

```
def touch_7(self, request):
```

```
    ansible_playbook = home_dir + '../ansible_file/touch_7.yml'
```

```
    bits = 'set -o pipefail;ansible-playbook {0}|tee -a /tmp/ansible.log'.format(ansible_
    playbook)
```

```
    (status, output) = commands.getstatusoutput('bash -c "{0}"'.format(bits))
```

```
    if status != 0:
```

```
        return Response({'msg': '{0} has not been touched'.format('ggg'), 'output':
        output})
```

```
    return Response({'msg': '{0} has beentouched'.format('ggg'), 'output': output})
```

我们来调用编写的 url 接口。

```
http://your_ip_address:8001/demo_1/demo_api/touch_7/
```

这样，我们就取得了所要的结果。不仅如此，我们的命令 rc 非 0 的错误输出仍然可以被使用，这样就省去很多时间，否则我们还要判断 output 的具体信息来分析。如果你用的是 API 的方式，那么就必须要进行这方面的处理，在这里我们可以省去这一步了。

```
HTTP 200 OK
```

```
Content-Type: application/json
```

```
Vary: Accept
```

```
Allow: GET, POST, HEAD, OPTIONS
```

```
{
```

```
  "msg": "ggg has not been touched",
```

```
  "output": "\nPLAY [localhost] *****
```

```
***** \n\nGATHERING FACTS *****
```

```
***** \nok: [127.0.0.1]\n\nTASK: [touch file] ****
```

```
***** \nfailed:
```

```
[127.0.0.1] => {"changed": true, "cmd": "touch /tmp_1/ggg ", "delta":
```

```
"0:00:00.002072", "end": "2016-05-03 22:05:35.973048", "rc": 1,
```

```
"start": "2016-05-03 22:05:35.970976"}\nstderr: touch: 无法创建"/tmp_1/
```

```
ggg": 没有那个文件或目录\n\nFATAL: all hosts have already failed -- aborting\
```

```
\n\nPLAY RECAP *****
```

```
***** \n          to retry, use: --limit @/home/deploy/touch_7.retry\
```

```
\n\n127.0.0.1          : ok=1    changed=0    unreachable=0
```

```
failed=1    \n"
```

```
}
```

通过以上的尝试，我们终于来到最关心的一步，把参数代入 Playbook 中去了，也就是第 8 个问题。创建一个最简单的 Playbook 的 YML 格式的文件，名称为 touch_8.yml。

```
- hosts: localhost

tasks:
- name: touch file
  shell: touch /tmp_1/{{filename}}
```

想要代入参数的话，一般通过 -e 来实现，比如要创建一个叫 sda 的文件，那么要传入的参数就是 -e '{"filename": "sda"}'。

```
@list_route(methods=['get', 'post'])
def touch_8(self, request):
    data = str(request.data).replace("u", '\\').replace("'", '\\\'')
    ansible_playbook = home_dir + "../ansible_file/touch_8.yml"
    bits = 'ansible-playbook {0} -e'.format(ansible_playbook) + ' "' + '{0}' + '".format(data)
    + '|tee -a /tmp/ansible.log'
    comm = 'bash -c "' + 'set -o pipefail;{0}'''.format(bits)
    (status, output) = commands.getstatusoutput(comm)
    if status != 0:
        return Response({'msg': '{0} has not been touched'.format('ggg'), 'output':
            output})
    return Response({'msg': '{0} has been touched'.format('ggg'), 'output': output})
```

在这里由于 Ansible 中使用的参数时会带来过多的单引号和双引号，我们需要提前对程序做处理，否则，将不能同时使用 bash -c 'set -o pipefail; xxx'，这点需要特别注意。

```
data = str(request.data).replace("u", '\\').replace("'", '\\\'')
```

我们最后来调用一次刚才的接口。

```
curl -X post http://your_ip_address:8001/demo_1/demo_api/touch_8/ -H "Content-Type: application/json" -d '{"filename": "/tmp1/sda"}'
```

我们执行后，得到以下运行结果：

```
PLAY [localhost] *****
```

```
GATHERING FACTS *****
```

```
ok: [127.0.0.1]
```

```
TASK: [touch file] *****
```

```
failed: [127.0.0.1] => {"changed": true, "cmd": "touch /tmp_1// tmp1/sda ",
"delta": "0:00:00.002124", "end": "2016-05-08 14:08:04.868095", "rc": 1,
"start": "2016-05-08 14:08:04.865971"}
```

```
stderr: touch: 无法创建 "/tmp_1// tmp1/sda": 没有那个文件或目录
```

```
FATAL: all hosts have already failed -- aborting
```

```
PLAY RECAP *****
```

```
to retry, use: --limit @/home/deploy/touch_8.retry
```

```
127.0.0.1          : ok=1    changed=0    unreachable=0    failed=1
```

至此，我们已经完成了所有的 Ansible 命令行调用的方式了，这样已经够用了，而且足够方便。如果想用 API 的方式调用，只要用第 12 章的 API 调用的类替换上述代码中的 `commands` 模块即可。

13.4 前端基础知识介绍

大家在阅读下一节内容之前，先去了解下 HTML、CSS、JavaScript 这 3 部分的内容，不需要深入了解，但要知道它们三者是怎么结合的，其中 HTML、CSS 读者们稍作了解即可。我们将会使用 Bootstrap3 替换大部分 HTML、CSS 的部分，借此来简化静态页面的编写。而对于 JavaScript，可能读者要多花点心思了，至少要知道原生 JavaScript 的一些基础语法规则，之后我们会使用 Angularjs 来简化 JavaScript 的编写。

13.4.1 HTML 和 CSS 简介

对于 Bootstrap3，大家可以通过 Google 或者 Baidu 搜索 Bootstrap 中文教程即可，里面有详细的介绍，在此不多介绍。但考虑到后面章节的需要，还是会简单介绍一下后面会用到的部分。

首先介绍一下 html 文件的基础结构。

```
<html lang="zh-CN">
  <head>
    <meta charset="utf-8">
    <title>Bootstrap3</title>
    <link href="css/bootstrap.min.css" rel="stylesheet">
  </head>
  <body>
    web 内容
    <script src="js/bootstrap.min.js"></script>
  </body>
</html>
```

首先要关心的是那些静态文件 CSS、JavaScript 的引用问题，原则是尽量把 CSS 文件的调用放到 `head` 这个标签里面。而为了加快 JavaScript 文件的读取速度，我们推荐把所有的 JavaScript 文件放在最后，即 `body` 标签的末尾。

我们需要知道一些 Bootstrap3 的基础知识：Bootstrap3 使用的是栅格类布局，它把页面的每一行分为 12 个栅格，当你用到其中一个栅格编写内容时，只需要在对应 HTML 标签中使用 `col-md-1` 就可以了。

例如:

```
<div class="col-md-1">.col-md-1</div>
```

我们后面用到 Bootstrap3 的内容不会很多, 主要以教学和介绍为主。我们列举说明一下稍后会用到的一些类, 请大家对这些稍做了解。

- col-md-*: 基础栅格类;
- col-md-offset-*: 栅格后退类;
- form-control: 输入美化类, 主要用于标签 input;
- control-label: 标签美化类, 主要用于标签 label;
- btn-primary: 按钮美化类, 主要用于标签 button。

13.4.2 JavaScript 简介

接下来, 我们来介绍 JavaScript 中最流行的框架 Angular.js。它以简单易用著称, 它最主要的特性还是双向数据绑定, 能帮助我们很简单地把 HTML 和 JavaScript 里的内容关联起来。在使用之前, 需要对 Angular.js 做一些调整。

- 1) 定义 Angular 的模块的名字, 在这里我们定义为 myApp。
- 2) 由于如果要在 Django 中使用 Angular.js, 我们要把 HTML 数据绑定部分的标记符从 {{}} 改为其他的标记符 (我们在这里改为 (())), 以避免与原来 Django 模板标签的冲突, 因为 Django 模板也使用 {{}}。
- 3) 在 Django 请求上默认加上 csrftoken 的认证, 这样可以避免很多问题, 这也是让使用 Django 的新手经常碰到且很头痛的问题。

为了解决上面 3 个问题, 请把下面的 JavaScript 代码导入 HTML 主文件中, 使之永久生效。请在 HTML 文件中用如下方式导入 main.js。

```
<script type="text/javascript" src="{% static 'js/angular-factory/ main_ctrl.js' %}"></script>
```

ansible_demo/root_demo/static_root/js/angular-factory/main.js 文件内容: var app = angular.module('myApp', []);

```
    app.config(function($interpolateProvider) {
        $interpolateProvider.startSymbol('{{{{';
        $interpolateProvider.endSymbol('}}}}');
    });
```

```
app.run(
```

```
    function($http) {
        $http.defaults.xsrfCookieName = 'csrftoken';
        $http.defaults.xsrfHeaderName = 'X-CSRFToken';
    });
```

我们之后会使用它的一些关键组件, 最好请大家做好准备。不过我们也将后面的一节, 结合例子来给大家介绍。

(1) HTML 组件

□ ng-model: 定义双向绑定数据的 HTML 部分;

□ ng-click: 当单击时, 执行 JavaScript 的动作。

(2) JavaScript 组件

□ \$scope: 定义双向绑定数据的 JavaScript 部分;

□ \$http: 传递给后端的 Python 程序的前端 API。

13.5 Ansible WebUI 界面开发

我们在 13.3 节中说了那么多关于 http 调用的接口问题, 而且在 13.4 节中也简单地介绍了需要了解的前端知识, 所有条件都具备了, 那么就开始编写一个简单的 html 调用我们的接口吧。

13.5.1 对接前端页面与 Ansible 的 Web 接口

为了让大家的思路更加清晰, 分成 4 步来完成我们对前端页面和 Web 接口的结合的介绍。

步骤 1: 需要在 `ansible_demo/demo_1/urls.py` 里添加我们网页访问的 url 定义。

定义访问的 url 为 `demo_static`, 加上我们在 `root_demo` 的主目录中定义的一级目录, 那么我们访问的 url 就应该是 `demo_1/demo_static/`, 加上我们的主机 IP 和端口, 那就应该是 `http://youripaddress:8001/demo_1/demo_static/`。在 `demo_1/urls.py` 添加我们的 url, 另外, 还需要指定的视图名称 `demo_static`, 如 (`url(r'^url 地址', 视图名称)`)。

```
urlpatterns += [
    url(r'^demo_static/', demo_static),
]
```

步骤 2: 在 `views.py` 里添加视图, 并为这个视图指定它对应的 html 文件。

我们把上面的 url 地址与视图名称 `demo_static` 这个方法对应起来, 在最后我们需要指定 html 文件 `demo_static.html`, 如 (`render(request, 'html 文件')`)。

```
@api_view(['GET', 'POST'])
def demo_static(request):
    if request.method == 'GET':
        return render(request, 'demo_static.html')
```

通过前两步我们很清晰地了解了 url 地址与 html 文件的关系, 这个我们在 13.2 节简单提过: url (访问地址) → views (视图方法) → html (html 文件位置)。

步骤 3: 编写 html。

我们来编写一个最简单的 html 文件, 根据 `views.py` 中的 `demo_static` 这个方法的指定,

编写的文件为 demo_static.html。Django 所有的定义的 html 都以 template 为根目录，所以应该编写的真正目录文件为 ansible_demo/root_demo/templates/demo_static.html。看以下 template 模板。

```
{% extends "normal/base.html" %}

{% load staticfiles %}
{% block title %}demo_static{% endblock %}

{% block content %}
{% include "demo/demo_static.html" %}
{% endblock %}

{% block chosen_js_file %}
<!-- end -->

{% endblock %}
```

大家是不是没有看到任何的 html 标签？我们还未满足编写的那个简单的页面条件，因为需要提前导入我们以后可能用到的一些 css 和 js 文件，可以通过 Django 的块级标签来导入 {% extends "normal/base.html" %}，之后就可以编辑真正的 html 文件 demo/demo_static.html 了。我们还是通过 include 标签导入 {% include "demo/demo_static.html" %}，来看下 demo_static.html。

```
<div class="col-md-12">
  <div class="row col-md-12">
    <br />
    <p>it's work</p>
  </div>
</div>
```

如果大家有兴趣可以自己编写 normal/base.html，笔者已经为大家导入 jQuery、Bootstrap、Angular.js 这些样式，如需要添加，请编写入其中。

接下来大家可以访问我们的网页查看结果，如出现 it works，那一个网页就定义完成了。

步骤 4：为 html 添加它的 JavaScript 控制文件。

现在来调用我们写的接口 /demo_1/demo_api/touch_2/，来完成在页面上调用 Ansible 执行的任务，我们主要使用 Angular.js 来完成，此处要处理 http（Ajax）的回调问题，Angular.js 提供了一个很好的方法 \$http，因为 /demo_1/demo_api/touch_2/ 这个接口只使用 \$http 的 get 方法即可，其回调功能主要在 .success 和 .error 的方法中。接下来，我们来看 JavaScript 文件。

```
app.controller('demo_test1_ctrl',function($scope, $http){
  $scope.touch_file = function() {
    alert('开始创建文件');
    $http.get('/demo_1/demo_api/touch_2/')
```

```

        .success(function(result){
            alert('创建文件成功');
            console.log(result)
        }).error(function(err){
            alert('创建文件失败');
            console.log(err)
        });
    });
});

```

如果我们要调用 JavaScript 的这个 controller，首先需要在 html 文件中引用这个 JavaScript 文件。即在 `ansible_demo/root_demo/templates/demo_angular.html` 文件中使用。

```

{% block chosen_js_file %}
    <script type="text/javascript" src="{% static 'js/angular-controller/demo_
ansible/demo_test1_ctrl.js' %}"></script>
{% endblock %}

```

之后在 html 文件的最高层必须声明 `ng-controller="demo_test1_ctrl"`，以此来调用这个叫作 `demo_test1_ctrl` 的 controller。即在 `ansible_demo/root_demo/templates/demo/demo_test1.html` 中使用以下形式：

```
<div class="col-md-12" ng-controller="demo_test1_ctrl">
```

我们来看完整的 html 文件。

```

<div class="col-md-12" ng-controller="demo_test1_ctrl">
    <div class="row col-md-12">
        <br />
        <button class="btn btn btn-primary col-md-offset-5" ng-click="touch_
file()">使用 ansible 创建文件 </button>
    </div>
</div>

```

我们现在就可以使用网页来调用我们的接口了。访问以下 url 地址：

```
http://youraddress:8001/demo_1/demo_angular/
```

我们得到的网页应该是一个简单的按钮，创建文件页面如图 13-1 所示。



图 13-1 创建文件页面

现在读者可以尝试自己单击下这个按钮，是不是得到了我们所要的结果了呢？

13.5.2 配置 Web 页面传参

以上的所有步骤的努力都是为了最后一步：传入我们的参数给前端页面。

1) 同样在 `urls.py` 和 `views.py` 中添加 url 路由并指定它的 html 文件, 我们把 url 定义为 `demo_ansible_with_vars`, 并指定对应的方法 `demo_ansible_with_vars`, 关联我们的 html 文件 `demo_ansible_with_vars.html`。这个文件我们之前已经介绍过了, 详见: `ansible_demo/demo_1/urls.py` 和 `ansible_demo/demo_1/views.py`。

2) 我们来修改 `demo_ansible_with_vars.html` 文件, 指定真正的 html 页面 `demo/demo_test2.html`, 并引入 JavaScript 文件, 并定义 controller 名分别为 `js/angular-controller/demo_ansible/demo_test2_ctrl.js` 和 `demo_test2_ctrl`。那么我们来看一下这两个文件 `demo/demo_test2.html`。

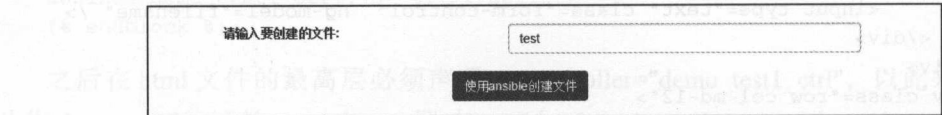
```
<div class="col-md-12" ng-controller="demo_test2_ctrl">
  <div class="form-group col-md-offset-3">
    <br/>
    <label class="col-md-3 control-label"> 请输入要创建的文件 :</label>
    <div class="col-md-4">
      <input type="text" class="form-control" ng-model="filename" />
    </div>
  </div>
  <div class="row col-md-12">
    <br />
    <button class="btn btn btn-primary col-md-offset-5" ng-click="touch_file(filename)">使用 ansible 创建文件 </button>
  </div>
  <br/>
  <div class="col-md-12">
    <p class="col-md-offset-3">{{ main_msg }}</p>
    <br/>
    <label class="col-md-offset-3 control-label" ng-if="ansible_log">ansible 日志 :</label>
    <br/>
    <textarea class="form-control" rows="20" ng-model="ansible_log" ng-if="ansible_log"></textarea>
  </div>
</div>
```

在这里我们传入的文件通过 `ng-model="filename"` 来定义, 把这个 `filename` 传入 JavaScript 的 `$scope.touch_file` 方法中, 把 `$http` 的方法改成传参的 `post` 方法, 并修改我们的接口为 `/demo_1/demo_api/touch_8/` 就可以了。我们再来看 JavaScript 文件:

```
app.controller('demo_test2_ctrl',function($scope, $http){
  $scope.touch_file = function(filename) {
    $scope.ansible_log = '';
    if(_.isEmpty(filename)){
      alert(' 输入不能为空 ');
      return;
    }
    $scope.main_msg = ' 开始创建文件 , 请耐心等待 ';
    data = {'filename': filename}
    $http.post('/demo_1/demo_api/touch_8/', data)
```

```
.success(function(result){
    alert(result['msg']);
    console.log(result)
    $scope.ansible_log = result['output'];
    $scope.main_msg = '';
}).error(function(err){
    console.log(err)
});
});
});
```

这样我们的工作基本已经完成了，我们来访问已经编写好的网页吧。
`http://youraddress:8001/demo_1/demo_ansible_with_vars/`
我们会看到这个页面，带参数的创建文件页面如图 13-2 所示。



我们来创建一个叫 test 的文件，填入空格中之后，单击“使用 ansible 创建文件”按钮即可。在运行的时候可能会花点时间，这个时间取决于 Ansible 命令运行消耗的时间，请耐心等待。之后读者就可以看到我们的最终结构页面了，页面运行结果如图 13-3 所示。

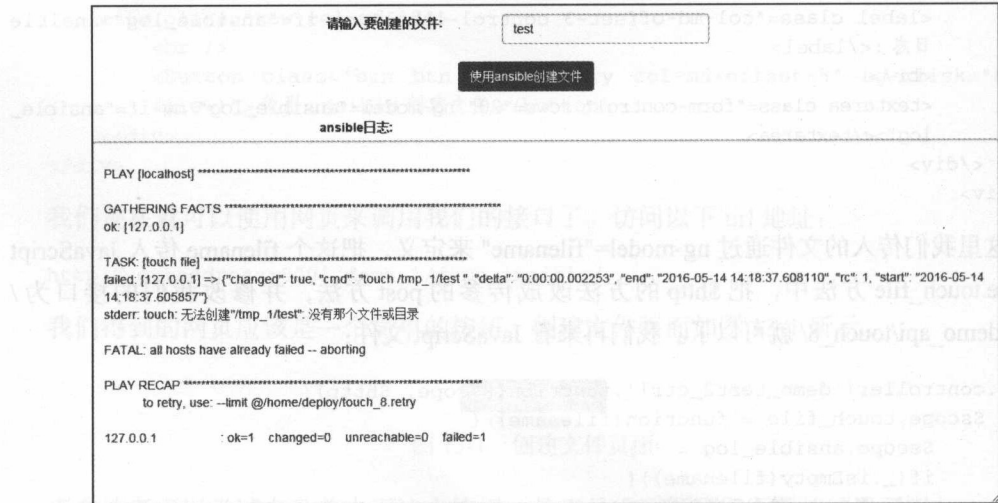


图 13-3 页面运行结果

接下来大家可以发挥自己的创造能力，编写更多的接口和与之配对的页面来完成一些功能。但由于还没有做权限控制，请大家不要放到互联网上去。

13.6 本章小结

在这章中，我们开始学习如何使用现有的工具和语言来编写属于自己的 Ansible 调用平台，主要通过现在最流行的几门语言和框架即 Django 作为后端语言，Angularjs+ Bootstrap3 作为前端。这样我们也可以寻求 Ansible Tower 以外的集中管理的页面方式来完成我们的工作了。

目前介绍的阶段虽然内容比较简单，但通过本章内容的学习，我们可以很清晰地了解到开发流程和各个语言之间的工作关系，虽然对于初学者比较困难，但笔者提供的开发方式和配置相对清晰，且是一一对应的，即一个页面对应一个 HTML+Angular 的一个 controller 的 js 文件，通过该 controller 文件调用后端 Django 的 API 来实现整个页面构建过程，通过切片式的编写，我们可以将这些页面最后重新拼成一个页面，这样也会大大降低我们的工作量，同时也有利于同事之间的分工合作。

在下一章，我们还会进一步深化学过的内容，将我们日常经常会遇到的问题和功能以实例的形式为读者展示。

Web 与 Ansible 结合的常用实例

在上一章中，我们学习了 Web 与 Ansible 结合使用的简单例子。本章我们将把重点放在读者日常可能使用到的例子，为大家尽可能多地打开设计的思路。

14.1 Web 方式管理 Ansible 的 Inventory

Ansible 的 hosts 文件在前面的章节中已经介绍过了，可能觉得没什么值得多说的，是与 Linux 机器的 hosts 文件属于同一类性质的文件，只是作为定义主机提供 Ansible 命令，指定目标主机来使用的。但是笔者想告诉大家的是，请一定要尽可能地运用 Ansible 对主机定义的特性，来制定尽可能完整又相对简洁的 hosts 文件，因为它将影响你后续程序设计的全局文件，请大家考虑好再对它进行规划。我们会将它的信息存入数据库，再反向生成该 hosts 文件，这样后续迁移到其他机器也相对容易一些。

14.1.1 重新定制 Ansible 的 Hosts 文件规则

对于 Ansible 的 hosts 文件管理，我们需要对它的格式做一定的规定，以更方便我们后续更容易在数据库层面上读取和存储，为我们后续的 Web 设计提供基石。

首先，我们制定 hosts 文件应该遵循两点规则：

- 1) 制定时请遵循简单明了的原则，避免产生歧义。
- 2) 最好不要有其他复杂的 Inventory 的定义，满足精确到一台机器即可。

对于上述 1)，大家可能觉得 hosts 文件写出以下方式就可以了：

```
[group-test1]
```



```
192.168.1.2
192.168.1.3
```

但随着设计的深入和复杂，这种简单的方式将不再适合设计的需要，根据我们上面的第 1 点规则，我们建议 Ansible 的 hosts 文件的写法最好是如下方式，因为它把所有最基础和必要的信息都写入文件中，避免了所有的歧义。

```
[group-test1]
test_test1 ansible_ssh_port=22 ansible_ssh_host=192.168.1.2
            ansible_ssh_user=deploy
test_test2 ansible_ssh_port=22 ansible_ssh_host=192.168.1.3
            ansible_ssh_user=deploy
```

以下设计也是可以的（用冒号来定义主机名），但是只适用于 Ansible 1.7.x 以上的版本，Ansible 2.0 以上版本将不再支持对此类文件的解析。

```
test_test1: ansible_ssh_port=22 ansible_ssh_host=192.168.1.2
            ansible_ssh_user=deploy
```

再举一个错误的例子，请大家也不要写成以下形式：

```
[group-test1]
192.168.1.2 ansible_ssh_port=22 ansible_ssh_user=deploy
192.168.1.3 ansible_ssh_port=22 ansible_ssh_user=deploy
```

这时因为上面那个 hosts 文件的两个 IP 并没有冲突，如果是下面的形式，第 2 台机器（尽管它的端口不一样）将永远不会被读取（如 `ansible 192.168.1.2 -a xxx`），如果读者把 hosts 文件写成下面形式的格式，那么端口为 222 的这台机器将无法被读取。

```
[group-test2]
192.168.1.2 ansible_ssh_port=22 ansible_ssh_user=deploy
192.168.1.2 ansible_ssh_port=222 ansible_ssh_user=deploy
```

所以请使用我们推荐的方式来规范你的配置文件，从而有效地避免 Ansible 识别主机的歧义。

对于规则的第 2 条的解释，保持一个文件的简洁性和主机定义的原子性，可以有效地为我们后续设计的多样和复杂性铺平道路，因为我们以后选择的机器和机器组，将会使用“:”进行选择 and 拼接（例如 `test_test3:test_test1`），虽然我们觉得这样每次调用 Playbook 写如此长的 host 定义会比较麻烦，但是如果交给给我们后续设计的程序来拼接，就一点也不麻烦了。

这样，大家可能没什么概念，大家之后可以设计成如下，这样我们就可以精确到主机节点，后续就可以设计更复杂的组合了，主机添加列表如图 14-1 所示。

14.1.2 使用 ConfigParser 解析并生成 Ansible Hosts 文件

在 14.1.1 节里我们介绍了如何定义 hosts 文件，并解释了这样定义的原因。接下来我们

要通过数据库取到的 JSON 数据来为 Ansible 文件产生下面的 hosts 列表。使用 JSON 格式的数据是因为我们后面会使用 JavaScript，而其交换数据是通过 JSON 格式来完成的。例如生成后为以下文件：

重新生成ansible hosts文件		添加服务器列表				
主机名称	组名	主机ip	用户名	端口	机器类型	操作
test1	group-test1	127.0.0.1	deploy	22	server	修改和查看 删除
test2	group-test1	127.0.0.1	deploy	22	server	修改和查看 删除
test3	group-test2	127.0.0.1	deploy	22	server	修改和查看 删除
test4	group-test3	127.0.0.1	user	22	server	修改和查看 删除

图 14-1 主机添加列表

```
[test-group1]
test1  ansible_ssh_port=22 ansible_ssh_host=127.0.0.1 ansible_ssh_user=deploy
test2  ansible_ssh_port=22 ansible_ssh_host=127.0.0.1 ansible_ssh_user=deploy

[test-group2]
test3  ansible_ssh_port=22 ansible_ssh_host=127.0.0.1 ansible_ssh_user=deploy
```

为了产生上面的 Ansible 的 hosts 主机信息，我们将使用 Python 的 ConfigParser 模块。但有一点需要注意，为了便于区分 Ansible 的 key 和 value，我们将统一使用两个空格为键值（如 test1 ansible_ssh_port=22）。但是由于原生的 ConfigParser 模块在保存文件后将自动把冒号和空格等解析成等号（如 test1= ansible_ssh_port=22），从而造成 Ansible 无法解析生成的 hosts 文件。所以为了避免键值标记的冲突（主要是等号），我们将对 ConfigParser 模块进行扩展。那么来编写 Part1，把 ConfigParser 模块修改为我们所能使用的模块 KconfigParser。

```
# !/usr/bin/env python
# -*- coding: utf-8 -*-

import json
import ConfigParser
class KConfigParser(ConfigParser.RawConfigParser):
    def write(self, fp):
        """ 解决 ConfigParser 的冒号、空格等被自动保存为等号而引起的后续解析问题 """
        if self._defaults:
            fp.write("[%s]\n" % DEFAULTSECT)
            for (key, value) in self._defaults.items():
                fp.write("%s %s\n" % (key, str(value).replace('\n', '\n\t')))
            fp.write("\n")
        for section in self._sections:
            fp.write("[%s]\n" % section)
            for (key, value) in self._sections[section].items():
                if key != "__name__":
                    fp.write("%s %s\n" %
```

```
(key, str(value).replace('\n', '\n\t')) # 使用两个空格取代 key
fp.write("\n")
```

接下来我们将会使用 `KconfigParser` 来替代原生的 `ConfigParser`，这里我们将用两个空格来替代等号或者冒号作为 `key`，而且经过测试，空格将通过 Ansible 所有版本的主机名识别。为了区别其他空格（如 `ansible_ssh_port=22 ansible_ssh_host=127.0.0.1`），我们改为用两个空格区分 `key` 和 `value`（如 `test1 ansible_ssh_port=22`），冒号将在 Ansible2+ 版本不被识别。我们来写如何生成自己的 `hosts` 文件的方法，来编写 Part2。

```
class Generate_ansible_hosts(object):
    def __init__(self, host_file):
        self.config = KconfigParser(allow_no_value=True)
        self.host_file = host_file

    def create_all_servers(self, items):
        for i in items:
            group = i['group']
            self.config.add_section(group)
            for j in i['items']:
                name = j['name']
                ssh_port = j['ssh_port']
                ssh_host = j['ssh_host']
                ssh_user = j['ssh_user']
                build = "ansible_ssh_port={0} ansible_ssh_host={1} ansible_ssh_
                    user={2}".format(
                        ssh_port, ssh_host, ssh_user)
                self.config.set(group, name, build)
            with open(self.host_file, 'wb') as configfile:
                self.config.write(configfile)
            return True
```

在使用我们的数据库前（程序取出后我们将其转化为 JSON 格式），我们首先使用一些假定的 JSON 数据来代替我们之后从数据库取出的数据，来完成编写测试工作。以下就是我们定义的 JSON 数据，并用来存入 `hosts` 文件。

```
[
    {
        "group": "group-name",          # 定义组名
        "items": [
            {
                "name": "host-name",     # 定义主机名
                "ssh_host": "host-ip",   # 定义主机 ip
                "ssh_port": host-port,    # 定义主机端口
                "ssh_user": "host-user"  # 定义信任用户
            }
        ]
    }
]
```

最后，那么我们来编写 Part3 制造的一些 JSON 数据，同时编写语句来测试我们上面的方法。

```
generate_hosts = Generate_ansible_hosts('/tmp/hosts') # 定义 Ansible 的 hosts 文件
data = [
    {
        "group": "test-group1",
        "items": [
            {
                "name": "test1",
                "ssh_host": "127.0.0.1",
                "ssh_port": 22,
                "ssh_user": "deploy"
            },
            {
                "name": "test2",
                "ssh_host": "127.0.0.1",
                "ssh_port": 22,
                "ssh_user": "deploy"
            }
        ]
    },
    {
        "group": "test-group2",
        "items": [
            {
                "name": "test3",
                "ssh_host": "127.0.0.1",
                "ssh_port": 22,
                "ssh_user": "deploy"
            }
        ]
    }
]

generate_hosts.create_all_servers(data) # 生成数据
```

这样，我们便编造好了存储的 JSON 格式数据了。之后，通过上述的 `generate_hosts.create_all_servers` 方法来生成 Ansible 的 hosts 文件即可。

14.1.3 使用数据库的存储数据生成的 Ansible Hosts 文件

上一节，我们已经设计好 JSON 的数据格式，那么我们就参照这个数据格式来编写数据库的字段。

我们简单地设计了字段：添加在文件在 `ansible_demo/demo_2/api/models.py` 中的内容如下：

```
class Ansible_Host(models.Model):
```



```

group = models.CharField(max_length=50, blank=True, default='') # 组名
name = models.CharField(max_length=50, blank=True, default='') # 主机别名
ssh_host = models.CharField(max_length=50, blank=True, default='') # 主机 ip
ssh_user = models.CharField(max_length=50, blank=True, default='') # 信任的用户名
ssh_port = models.CharField(max_length=50, blank=True, default='') # 主机端口
server_type = models.CharField(max_length=100, blank=True, default='') # 主机类型
commit = models.TextField(blank=True, null=True) # 备注

```

在设计好 models.py 的文件后，我们就可以生成 demo_2 数据库的 Ansible_Host 表结构。我们可以使用以下命令来完成：

```

cd ansible_demo
./manage.py makemigrations demo_2
./manage.py migrate demo_2

```

我们要编写接口来生成 Hosts 文件，这时我们上面编写的生成 Hosts 文件的方法也可以被我们的接口调用了。

之后，我们就在 http://youripaddress:yourport/demo_2/ansible_host_api/ 里输入一些数据，如图 14-2 所示。

	Raw data	HTML form
Group	group-test1	<input type="text" value="group-test1"/>
Name	test1	<input type="text" value="test1"/>
Ssh host	127.0.0.1	<input type="text" value="127.0.0.1"/>
Ssh user	deploy	<input type="text" value="deploy"/>
Ssh port	22	<input type="text" value="22"/>
Server type	server	<input type="text" value="server"/>
Comment	test	<input type="text" value="test"/>

POST

图 14-2 添加 host 数据

我们先任意添加一些测试数据，如图 14-3 所示。

有了这些测试数据后，我们就可以着手编写生成 Ansible 的 hosts 接口了，通过使用上面编写的 Generate_ansible_hosts 方法即可完成生成工作。

ansible_demo/demo_2/api/api_demo_2.py 文件如下：

```

def generate_hosts(self):
    data = Ansible_Host.objects.all().values()
    s_data = [{'group': group, 'items': list(items)} for group, items in itertools.
groupby(data, lambda x: x['group'])]

```

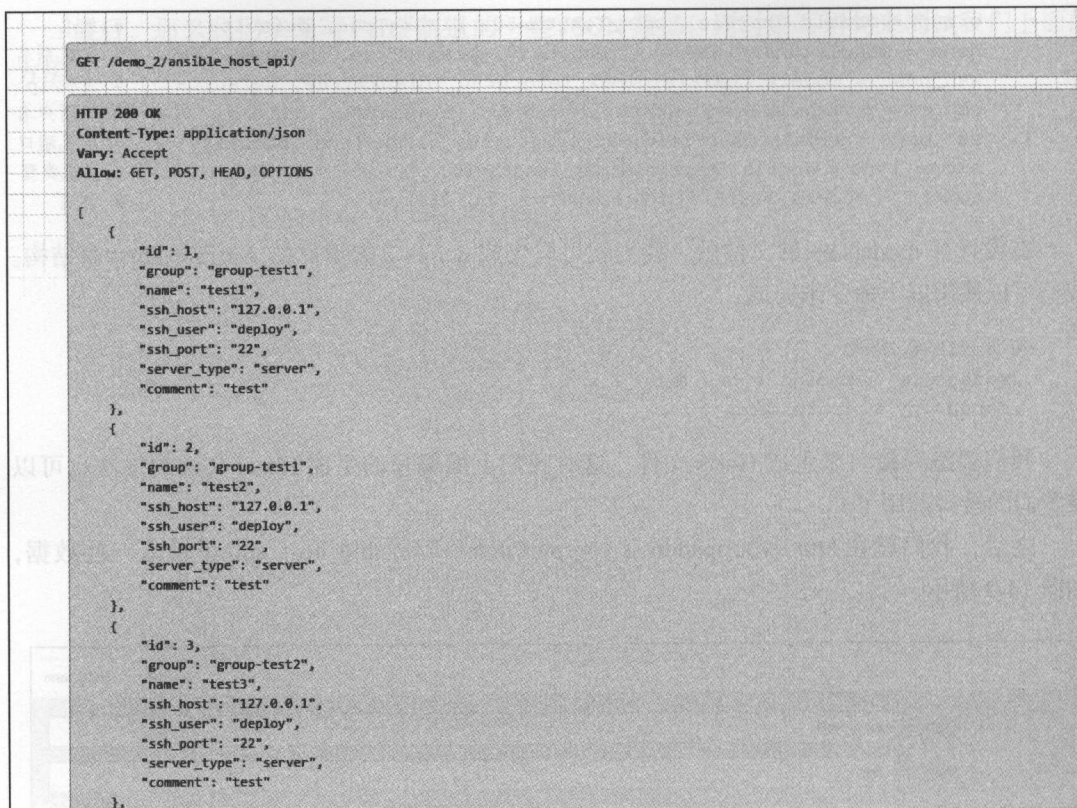


图 14-3 添加测试数据

```

generate_hosts = Generate_ansible_hosts('/tmp/hosts')
try:
    generate_hosts.create_all_servers(s_data)
    msg = 'hosts has been create'
    flag = True
except:
    msg = 'hosts has not been create'
    flag = False
return msg, flag

@list_route(methods=['get', 'post'])
def create_ansible_hosts(self, request):
    msg, flag = self.generate_hosts()
    return Response({'msg': msg, 'flag': flag})

```

请真正使用时应把 `generate_hosts = Generate_ansible_hosts('/tmp/hosts')` 更改为 `generate_hosts = Generate_ansible_hosts('/etc/ansible/hosts')`，并给这个文件对应地写入权限。

所有工作完成后，我们来访问一下接口，并检查是否生成了的 `hosts` 文件：`http://youripaddress:yourport/demo_2/demo2_api/create_ansible_hosts/`。

14.1.4 通过页面来生成 Hosts 文件

接下来就要构建页面了，因为这是第一次比较完整地构建应用界面，笔者将拆分得比较细，将以 tag 标签的方式来说明问题。由于篇幅有限，不能将所有代码放上，我们只展示各个 tag 之间不同的代码。接下来我们就来循序渐进地完成所有功能。

5 个 Tags 问题标签组如下。

- 最简单的页面来展示我们之前输入到 hosts 数据的页面（问题标签：deploy）。
- 数据库中的主机信息自动生成的 Ansible 的 hosts（问题标签：create）。
- 在页面上添加一个主机，并记录数据库同时生成 hosts 文件（问题标签：add）。
- 在页面上删除一个主机，并记录数据库同时生成 hosts 文件（问题标签：delete）。
- 在页面上更改一个主机，并记录数据库同时生成 hosts 文件（问题标签：modify）。

编写的逻辑流程如图 14-4 所示。

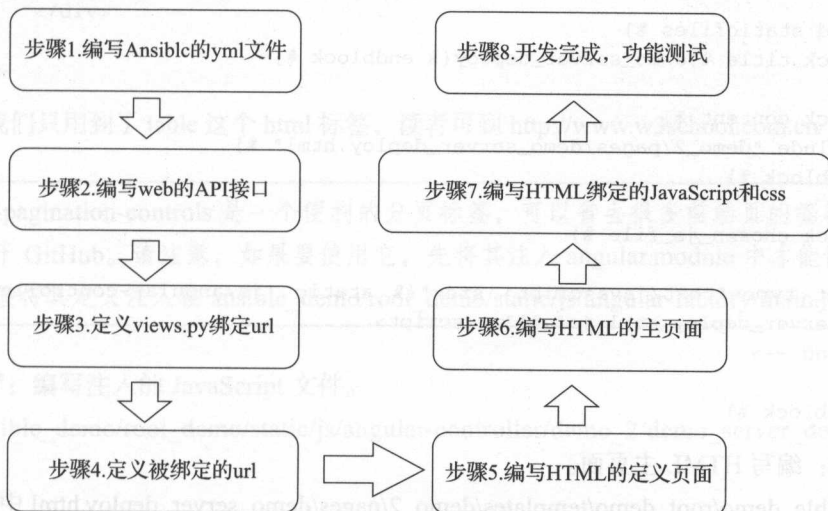


图 14-4 页面编写流程

注意 第 1 步和第 2 步如果只是编写展示页面，可以跳过。

现在，我们就按照上面的流程图来快速形成页面。

1) 完成展示我们之前输入到 hosts 数据的页面（问题标签：deploy）。

我们下面所说的流程，都参照图 14-4 页面编写流程。

步骤 1：该示例未用到 YAML 文件，跳过。

步骤 2：该示例未用到 API 接口，跳过。

步骤 3：定义绑定 url。

在 `ansible_demo/demo_2/views.py` 中添加:

```
@api_view(['GET', 'POST'])
def demo_server_deploy(request):
    if request.method == 'GET':
        return render(request, 'demo_2/defines/demo_server_deploy.html')
```

步骤 4: 定义被绑定 url。

在 `ansible_demo/demo_2/urls.py` 中添加:

```
url(r'^demo_server_deploy/', demo_server_deploy),
```

步骤 5: 定义 HTML 基础页面。

在 `ansible_demo/root_demo/templates/demo_2/defines/demo_server_deploy.html` 中添加:

```
{% extends "demo_2/base/base.html" %}

{% load staticfiles %}
{% block title %}demo_server_deploy{% endblock %}

{% block content %}
{% include "demo_2/pages/demo_server_deploy.html" %}
{% endblock %}

{% block chosen_js_file %}

<script type="text/javascript" src="{% static 'js/angular-controller/demo_2/
demo_server_deploy_ctrl.js' %}"></script>
<!-- end -->

{% endblock %}
```

步骤 6: 编写 HTML 主页面。

在 `ansible_demo/root_demo/templates/demo_2/pages/demo_server_deploy.html` 中添加:

```
<div ng-controller='demo_server_deploy_ctrl' class="col-md-12">
    <div class="row col-md-12">
        <div class="col-md-12">
            <table class="table table-bordered table-striped table-hover">
                <thead>
                    <tr class="info">
                        <th class="col-md-2 text-muted">主机名称 </th>
                        <th class="col-md-2 text-muted">组名 </th>
                        <th class="col-md-1 text-muted">主机 ip </th>
                        <th class="col-md-1 text-muted">用户名 </th>
                        <th class="col-md-1 text-muted">端口 </th>
                        <th class="col-md-2 text-muted">机器类型 </th>
                    </tr>
                </thead>
                <tbody dir-paginate="j in servers_list_all | itemsPerPage: 10
```




```

        track by $index" class="success" pagination-id="servers_list">
        <th class="col-md-2 text-muted">{{(j.name)}}</th>
        <th class="col-md-2 text-muted">{{(j.group)}}</th>
        <th class="col-md-1 text-muted">{{(j.ssh_host)}}</th>
        <th class="col-md-1 text-muted">{{(j.ssh_user)}}</th>
        <th class="col-md-1 text-muted">{{(j.ssh_port)}}</th>
        <th class="col-md-2 text-muted">{{(j.server_type)}}</th>
    </tbody>
</table>
</div>
</div>

<div class="col-md-12">
    <div class="col-md-offset-5">
        <dir-pagination-controls pagination-id="servers_list"></dir-
        pagination-controls>
    </div>
</div>
</div>

```

这里我们只用到了 table 这个 html 标签，读者可到 <http://www.w3school.com.cn/> 自行了解。

 **注意** dir-pagination-controls 是一个便利的分页标签，可以省去很多前端页的编写工作，来源于 GitHub。请注意，如果要使用它，先将其注入 angular.module 中才能使用。笔者已经将其定义注入在 ansible_demo/root_demo/static/js/angular-factory/main.js 中了。

步骤 7：编写注入的 JavaScript 文件。

在 ansible_demo/root_demo/static/js/angular-controller/demo_2/demo_server_deploy_ctrl.js 中添加：

```

app.controller('demo_server_deploy_ctrl',function($scope, $http){
    get_all_data = function() {
        $http.get('/demo_2/ansible_host_api/') # 通过该接口获取数据，读者可自行访问
        .success(function (res) {
            $scope.servers_list_all = res;
        })
    }
}

get_all_data()
});

```

步骤 8：完成编写并测试。

这样我们就完成了展示 Ansible 的 hosts 文件的页面，现在来访问它。

http://youraddress:yourport/demo_2/demo_server_deploy/

2) 数据库中的主机信息自动生成的 Ansible 的 hosts (问题标签: create)

步骤 1: 该示例未用到 YML 文件, 跳过。

步骤 2: Web 生成 hosts 文件的 API 接口已经在上节完成, 跳过。

步骤 3: 定义绑定 url。

在 `ansible_demo/demo_2/views.py` 中添加:

```
@api_view(['GET', 'POST'])
def demo_server_create(request):
    if request.method == 'GET':
        return render(request, 'demo_2/definesdemo_server_create.html')
```

步骤 4: 定义被绑定 url。

在 `ansible_demo/demo_2/urls.py` 中添加:

```
url(r'^demo_server_create/', demo_server_create),
```

步骤 5: 定义 HTML 基础页面。

在 `ansible_demo/root_demo/templates/demo_2/defines/ demo_server_create.html` 和 之前 `deploy` 问题标签的页面修改以下项:

```
{% block title %}demo_server_create{% endblock %}
.....
{% include "demo_2/pages/demo_server_create.html" %}
.....
<script type="text/javascript" src="{% static 'js/angular-controller/demo_2/
demo_server_create_ctrl.js' %}"></script>
.....
```

步骤 6: 编写 HTML 主页面。

`ansible_demo/root_demo/templates/demo_2/pages/demo_server_create.html` 和 `deploy` 页面相比添加了以下代码:

```
<div class="col-md-3">
    <button class="btn btn btn-primary col-md-offset-5" ng-click="generate_new_
    ansible_host_func()">重新生成 ansible hosts 文件</button>
</div>
```



注意 `generate_new_ansible_host_func()` 的 JavaScript 的方法, 我们会调用之前用 Python 生成 hosts 的方法。

步骤 7: 编写注入的 JavaScript 文件。

`ansible_demo/root_demo/static/js/angular-controller/demo_2/demo_server_create_ctrl.js` 与之前 `deploy` 的 JavaScript 文件相比添加了以下代码:

```

$scope.generate_new_ansible_host_func = function() {
    $http.get('/demo_2/demo2_api/create_ansible_hosts/')
        .success(function (res) {
            alert('生成 hosts 成功')
        }).error(function (res) {
            alert('生成 hosts 失败')
        })
}

```

步骤 8: 完成编写并测试。

这样我们就完成了添加主机的 hosts 文件的页面，现在来访问它。

`http://youraddress:yourport/demo_2/demo_server_add/`

3) 在页面上删除一个主机，并记录数据库同时生成 hosts 文件 (问题标签: delete)。

步骤 1: 该示例未用到 YML 文件，跳过。

步骤 2: 已编写过 API 接口，所以跳过。

步骤 3: 定义绑定 url。

在 `ansible_demo/demo_2/views.py` 中添加:

```

@api_view(['GET', 'POST'])
def demo_server_delete (request):
    if request.method == 'GET':
        return render(request, 'demo_2/defines/demo_server_delete.html')

```

步骤 4: 定义被绑定 url。

在 `ansible_demo/demo_2/urls.py` 中添加:

```
url(r'^demo_server_delete/', demo_server_delete),
```

步骤 5: 定义 HTML 基础页面。

在 `ansible_demo/root_demo/templates/demo_2/defines/ demo_server_delete.html` 和之前 add 的页面中修改这些项:

```

{% block title %} demo_server_delete{% endblock %}
.....
{% include "demo_2/pages/demo_server_delete.html" %}
.....
<script type="text/javascript" src="{% static 'js/angular-controller/demo_2/
demo_server_delete_ctrl.js' %}"></script>
.....

```

步骤 6: 编写 HTML 主页面。

`ansible_demo/root_demo/templates/demo/demo_server_create.html` 和 create 页面相比添加以下代码:

```
<th class="col-md-3 text-muted"> 操作 </th>
```

```

.....
<th class="col-md-3 text-muted">
  <button class="btn btn-primary btn-offset" data-toggle="modal" data-
    target="# servers_operate" ng-click="create_tag('remove', j)">删除</button>
</th>
.....
<h3 class="modal-title text-center text-primary" ng-if="operate_type ==
  'remove'">删除节点</h3>
.....
<button type="button" class="btn btn-primary" data-dismiss="modal" ng-
  click="generate_ansible_host_func()" ng-if="operate_type == 'remove'">确认删除
</button>

```



注意 这里我们用到了 Angularjs 中的 ng-if 的类，如果条件成立。则该标签显示。在这里如果 operate_type == 'remove' 成立，显示运用这个类的标签。

步骤 7：编写注入的 JavaScript 文件。

在 ansible_demo/root_demo/static/js/angular-controller/demo_2/demo_server_add_ctrl.js 与之前 create 的 JavaScript 文件相比添加和修改以下内究：

Part1：单击任意修改的行按钮后，我们都会把该行的值重新定义给新的变量，并为每个操作打上标签（remove、create、update 等），这样我们可以把所有逻辑集成在下面的函数中。

```

$scope.create_tag = function(tag, ins){ #
  $scope.operate_type = tag;           # if(tag == 'remove'){
    $scope.remove_id = ins.id;
    $scope.group = ins.group,
    $scope.name = ins.name,
    $scope.ssh_host = ins.ssh_host,
    $scope.ssh_user = ins.ssh_user,
    $scope.ssh_port = ins.ssh_user,
    $scope.server_type = ins.server_type,
    $scope.comment = ins.comment
  }
}

```

Part2：创建新数据库记录主机的 tag、create。

```

$scope.generate_ansible_host_func = function() {
  $scope.save_wait = true;
  if($scope.operate_type == 'create') {
    data = {'group': $scope.group,
      'name': $scope.name,
      'ssh_host': $scope.ssh_host,
      'ssh_user': $scope.ssh_user,
      'ssh_port': $scope.ssh_port,
      'server_type': $scope.server_type,

```



```

        'comment': $scope.comment
    }
    $http.post('/demo_2/demo2_api/create_ansible_hosts_add/', data)
    .success(function (res) {
        if (res['flag'] == true) {
            alert('生成 hosts 成功')
            get_all_data()
        } else {
            alert('生成 hosts 失败')
        }
        $scope.save_wait = '';
    }).error(function (res) {
        alert('生成 hosts 失败')
        $scope.save_wait = '';
    })
}

```

Part3: 删除数据库记录主机的 tag、remove。

```

if($scope.operate_type == 'remove') {
    data = {'remove_id': $scope.id};
    $http.post('/demo_2/demo2_api/create_ansible_hosts_delete/', data)
    .success(function (res) {
        if (res['flag'] == true) {
            alert('生成 hosts 成功')
            get_all_data()
        } else {
            alert('生成 hosts 失败')
        }
        $scope.save_wait = '';
    }).error(function (res) {
        alert('生成 hosts 失败')
        $scope.save_wait = '';
    })
}
}

```

步骤 8: 完成编写并测试。

我们就完成了删除主机的 hosts 文件的页面，现在来访问它。

http://youraddress:yourport/demo_2/demo_server_delete/

4) 在页面上更改一个主机，并记录数据库同时生成 hosts 文件 (问题标签: modify)。

步骤 1: 该示例未用到 YML 文件，跳过。

步骤 2: 已编写过 API 接口，所以跳过。

步骤 3: 定义绑定 url。

在 `ansible_demo/demo_2/views.py` 中添加:

```
@api_view(['GET', 'POST'])
```

```
def demo_server_modify (request):
    if request.method == 'GET':
        return render(request, 'demo_2/defines/demo_server_modify.html')
```

步骤 4: 定义被绑定 url。

在 `ansible_demo/demo_2/urls.py` 中添加:

```
url(r'^demo_server_modify/', demo_server_modify),
```

步骤 5: 定义 HTML 基础页面。

`ansible_demo/root_demo/templates/demo_2/defines/demo_server_modify.html` 和 之前 `delete` 的页面中修改这些项:

```
{% block title %}demo_server_modify {% endblock %}
.....
{% include "demo_2/pages/demo_server_modify.html" %}
.....
<script type="text/javascript" src="{% static 'js/angular-controller/demo_2/
demo_server_modify_ctrl.js' %}"></script>
.....
```

步骤 6: 编写 HTML 主页面。

`ansible_demo/root_demo/templates/demo_2/pages/demo_server_modify.html` 和 `delete` 页面相比添加以下代码:

```
<h3 class="modal-title text-center text-primary" ng-if="operate_type ==
'modify'">更新节点 </h3>
.....
<button type="button" class="btn btn-primary" ng-click="generate_ansible_host_
func()" ng-if="operate_type == 'remove'"> 确认删除 </button>
```

步骤 7: 编写注入的 JavaScript 文件。

`ansible_demo/root_demo/static/js/angular-controller/demo_2/demo_server_modify_ctrl.js` 与之前 `deploy` 的 JavaScript 文件相比添加以下代码:

```
if($scope.operate_type == 'modify') {
    data = {'id': $scope.id,
        'group': $scope.group,
        'name': $scope.name,
        'ssh_host': $scope.ssh_host,
        'ssh_user': $scope.ssh_user,
        'ssh_port': $scope.ssh_port,
        'server_type': $scope.server_type,
        'comment': $scope.comment
    }
    $http.post('/demo_2/demo2_api/create_ansible_hosts_modify/', data)
    .success(function (res) {
        if (res['flag'] == true) {
```

```

        alert('生成 hosts 成功')
        get_all_data()
    } else {
        alert('生成 hosts 失败')
    }
    $scope.save_wait = '';
  }).error(function (res) {
    alert('生成 hosts 失败')
    $scope.save_wait = '';
  })
}

```

步骤 8: 完成编写并测试。

我们就完成了修改主机的 hosts 文件的页面，现在来访问它。

http://youraddress:yourport/demo_2/demo_server_modify/

14.2 使用 celery 后台执行任务

celery 是 Python 的一种后台任务管理的软件，相当于我们将命令放在后台执行，只要知道该任务的 id，就可以随时取消任务。

14.2.1 为什么要使用 celery

为什么要使用 celery？我们使用之前的方法来调用命令不是很好吗？但是，你可以试试如果使用页面调用接口时，断网或者不小心刷新了一下网页，命令还会继续进行下去吗？答案是：将会全部终止，如果我们要执行一个很长的任务，比如编译安装一个模块，时间又比较长，如果这样的事情发生了，我们的命令就不知道进行到什么地方了，特别是一些不可逆的操作，将会给我们带来一些严重的后果。举个例子，在更新版本时突然接到撤销该更新的指令的时候，如果只是在做更新前的准备，我们本来可以终止的，但如果我们没有使用后台任务 celery，就无法取消该操作。

14.2.2 使用 celery 的前期准备

在使用前，我们需要安装 Redis 来完成 celery 任务存储和结果存储，因为 celery 会使用 Redis 来记录你的结果，用来中断你的操作。读者可以使用其他软件来代替 Redis，比如 RabbitMQ，或者数据库。但 Redis 我们后期还可以作为缓存使用，所以在这里就直接使用它了。

首先在挑选 Redis 版本时，请安装可以设置键值的 Redis 版本，大于 3.0 的版本是没问题的，请读者自行安装，这里就不赘述了。

然后进入你的虚拟环境，即 source 你对应目录的 activate，如果忘记可以翻看第 13 章的内容。之后 pip 安装一下 Python 包。

```
pip install celery django-celery django-redis
```

ansible_demo/root_demo/dev_settings.py 中添加:

```
DEV_INSTALLED_APPS = (
    .....
    'djcelery',
)
```

在命令行执行创建 celery 对应的数据结构。

```
cd ansible_demo
./manage.py migrate
```

最后, 在 Django 的配置文件中添加 celery 的一些配置。

在 ansible_demo/root_demo/local_settings.py 中添加:

```
import djcelery
djcelery.setup_loader()
BROKER_URL = 'redis://localhost:6379/3'
CELERY_RESULT_BACKEND = 'redis://localhost:6379/3'
CELERY_ACCEPT_CONTENT = ['json']
CELERY_TASK_SERIALIZER = 'json'
CELERY_RESULT_SERIALIZER = 'json'
CELERY_DEFAULT_QUEUE = "default"

CELERY_TRACK_STARTED = True
```




注意 对于上述的 `BROKER_URL` 和 `CELERY_RESULT_BACKEND`, 笔者建议使用同一个 Redis 的键值库 (这里统一使用 3 库), 对于性能比较差的机器来说, 可以避免启动 celery 任务失败的情况。另外对于 `CELERY_TRACK_STARTED` 来说, 如果不设定它为 `True` 的话, 只要任务结束前 (无论是否存在该任务), 任务状态都是 `PENDING`, 因为开启的 celery 端口有限, 超过数量的任务也会排队, 任务状态也是 `PENDING`, 这样我们更加无从判断任务何时开始。反之, 我们设 `CELERY_TRACK_STARTED = True` 时, 当任务开始后, 任务状态会变成 `STARTED`。

现在可以启动 celery 了。

```
cd ansible_demo
./manage.py celery worker -c 10 -autoreload
```

参数说明:

- ❑ `-c` 表示启动 10 个通道, 即一次最多运行 10 个任务, 其他任务将会自动排队。
- ❑ `-autoreload` 表示修改后自动重启上面的 celery 命令来载入新的任务。

 **注意** 对于 `--autoreload` 有大家需要注意，除非你把任务写在 Django 的 app（如 `demo_1`、`demo_2` 目录下）的 `tasks.py` 下时，你使用 `celery` 类修改了任务方法后，会自动重载生效外，其他文件不会自动重载任务，新加的任务也就不会自动生效。所以一定要重新启动上面那条 `celery` 命令。但由于 `Ctrl-C` 组合键不一定能杀死所有 `celery` 进程，请写一个脚本再重启时使用 `kill` 来杀死。可以参考 `ansible_demo/start_celery.sh` 自己写一个。

`ansible_demo/start_celery.sh` 脚本如下：

```
process='ps -ef |grep -E "./manage.py celery worker -c 10 --autoreload"|awk
'{print $2}''
redis-cli config set stop-writes-on-bgsave-error no
for i in $process;
do
    if [ $i -eq 1 ]; then
        echo "do nothing"
    else
        kill -9 $i
    fi
done
./manage.py celery purge -f;./manage.py celery worker -c 10 --autoreload
```

再强调一下，对于任务管理软件 `celery`，请一定要注意：关闭它时使用上面的 `shell` 脚本使其正常关闭。


14.2.3 使用 `celery` 开始任务

14.2.2 节成功地运行了 `Django-celery`，我们就可以使用 `celery` 写一个简单的任务来调用 `Playbook`。在这里，我们简单地编写了两个 `Playbook`，使用 `sleep 20` 来模拟较长时间的操作。

第 1 个 `sleep 20` 来模拟较长时间的 `Playbook` 文件 `ansible_file/long_cmd_1.yml`。

```
- hosts: localhost

tasks:
  - name: echo num
    shell: bash -c "echo 1 > /tmp/long.log;sleep 20"
```

 **注意** 另一个文件 `ansible_file/long_cmd_2.yml` 和上述文本内容是一样的，只不过把 `echo 1` 改成 `echo 2`。

现在，有了 `YML` 文件，我们就要来编写 `celery` 的任务方法 `long_ansible_bg`。要把该方法变成 `celery` 的后台任务，需要给这个方法加上装饰器 `@task.throws=(Terminated,)` 这个参

数方便后面更好地取消这个任务，可以避免一些问题。

接下来，我们编写了一个后台任务 `long_ansible_bg`，来使得 Playbook 在后台运行。

使用 `celery` 完成后台执行程序文件 `ansible_demo/demo_2/tasks.py`。

```
# !/usr/bin/env python
# -*- coding: utf-8 -*-
from celery import task
from billiard.exceptions import Terminated
import commands
import os

home_dir = os.path.abspath('.') + '/'

def long_ansible_common(yml_file):
    ansible_playbook = home_dir + '../ansible_file/{0}'.format(yml_file)
    print ansible_playbook
    bits = 'set -o pipefail;ansible-playbook {0}|tee -a /tmp/ansible_long.log'.format(ansible_playbook)
    (status, output) = commands.getstatusoutput('bash -c "{0}"'.format(bits))

@task(throws=(Terminated,))
def long_ansible_bg(data):
    long_ansible_common('long_cmd_1.yml')    # 执行 ansible 的 yml
    long_ansible_common('long_cmd_2.yml')
    return {'msg': 'long ansible cmd has been executed'}
```

我们已经完成了 `celery` 后台启动 Ansible 的 YML 的方法，最后只要在我们的 API 上调用这个方法就可以了，见 `ansible_demo/demo_2/api/api_demo_2.py` 文件。

首先，先导入上面的 `celery` 的 `task` 方法。

```
from demo_2.tasks import long_ansible_bg
```

使用 `long_ansible_bg` 方法。

```
@list_route(methods=['get', 'post'])
def long_ansible_background_cmd(self, request):
    data = request.data
    s1 = long_ansible_bg.s(data)
    res = s1.delay()
    return Response({'task_id': res.task_id})
```

如上返回结果，我们一定要返回任务的 `task_id`，因为我们后续需要用这个 `task_id` 来看这个任务的状态或终止该任务。希望读者最好用 Redis 或者数据库来记录这个 `id`，以免刷新网页后丢失该 `task_id`。



注意 我们虽然完成了所有的编写过程，但需要提醒的是：不是所有人都喜欢在 `tasks.py` 里面编写的（有些人喜欢在 `tasks.py` 里 `import` 自己的任务，这样保存文件后任务将不会

自动注册到 celery 的 task 中), 为了后续可能因忽略这个问题而引起的排错问题, 现在在最好还是养成习惯, 每次编写好 celery 的任务后, 先重启下 celery 来载入新写的或修改的任务。

下面来访问我们的接口。

`http://myaddress:myport/demo_2/demo2_api/long_ansible_background_cmd/`

我们不会等 40s (2 个 Ansible 命令各 20s), 会很快得到结果, 因为 Ansible 的所有任务都放在后台了, 所以我们很快得到执行后台任务后返回的 task_id, 如图 14-5 所示。

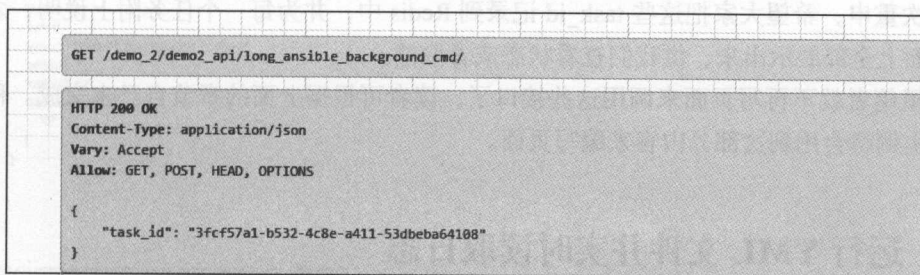


图 14-5 执行后台任务后返回的 task_id

我们通过 celery 完成了对任务的执行和编号, 而这个取得的任务编号 task_id, 我们将在后续很多地方用到, 所以对我们非常重要。

14.2.4 使用 celery 取消正在进行的任务

我们现在已经学会如果把任务放在后台了, 现在我们要学习怎么取消命令。

这里要用到 celery.task.control 中的 revoke 方法, 上面记录的 task_id 页面被要求作为参数传入这个 revoke 方法中, 来确定是哪一个任务。

还有一个方法也比较有用, 就是 celery.result 的 AsyncResult, 我们可以通过传入 task_id 给这个方法取得这个任务到底进行到什么程度的信息。我们主要会用到 status (任务成功、失败或者挂起) 和 result (任务的结果)。

我们新写一个 API 来使用这些方法取消任务。

```

@list_route(methods=['get', 'post'])
def long_ansible_revoke(self, request):
    data = request.data
    task_id = data['task_id']
    res = AsyncResult(task_id)
    revoke(task_id, terminate=True, signal='SIGKILL')
    return Response({'msg': task_id + ' has been revoked'})

```

我们现在来尝试让 long_ansible_background_cmd 只运行第一个 yml, 之后马上取消这个

任务来完成任务的中断测试。

我们再次运行第一个接口。

```
http://myaddress:myport/demo_2/demo2_api/long_ansible_background_cmd/
```

请在 20s 的时间里使用下面的方法来取消上面那个后台任务，请替换下面的 `#task_id#`：

```
curl -X post http://your_ip_address:yourport/demo_2/demo2_api/ long_ansible_revoke/ -H "Content-Type: application/json" -d '{"task_id": "# task_id#"}'
```

可以查看 `/tmp/long.log` 文件是不是在 40 秒过后仍是 1 而不是 2，如果是 1 的话，则说明任务取消成功。

再次重申，希望大家把这些 `task_id` 记录到 Redis 中，并为每一个任务附上说明，之后可以在页面上全部展示出来，供我们查看状态或者取消。

这里笔者就不再写页面来调用这些接口了，读者可根据上面的章节自己来完成。稍后几节中的实例将会用到这部分内容来编写页面。

14.3 运行 YAML 文件并实时读取日志

在我们的实例中应该比较迫切用到这个应用，如果在页面上不能获取 Ansible 的实时日志，就算我们用 Ansible 的 API 取得最后的 JSON 结果，我们也不能判断该任务到底哪里出了问题。再则一个任务会执行很长时间，如果我们没实时读取日志的话，就不能根据日志情况来中断任务，这也不符合我们的设计要求。所以，对于无须取得 JSON 结果内容，而只关心 YAML 文件运行时是否报错的需求，就不使用 Ansible 的 API 了，而是全部使用命令行的方式，并使用 `tee` 来实时写日志到临时文件。

所以，我们要完成的任务名称就将是：同步实时读取 Ansible 日志。

我们需要完成几个目标：

- 1) 任务开始时能显示任务状态；
- 2) 可以在任意时间结束任务；
- 3) 实时读取日志；
- 4) 随着日志的增加，日志能够在浏览器自动往下移动，无需手动移动滚动条。

我们来解释要完成以上功能的意义：

当我们需要执行一个耗时很长的任务时，如果只是调用 API 的话，它只返回一串结果的 JSON 状态，如果出现错误，你就很难判断是在哪里出错。再则如果你使用插件来取得 log，它也是要等待整个 Ansible 的 API 调用结束才会返回结果，中间我们可能根本不知道它运行到什么程度了，这样又违反了 Ansible 的设计初衷。若日志不能实时读取，那么使用命令行来执行更好。另外我们还需要解决一个日志滚动显示的问题，当左侧滚动条移到最底部时日志自动向下移动，在其他位置则保持日志不动，相当于完成一个 Google 浏览器的 console 界


```
@task(throws=(Terminated,))
def long_ansible_read_log(data):
    print 'start job'
    for i in xrange(10):
        print str(i) + ': {}'.format(data['yaml_file'])
        long_ansible_common(data['yaml_file'], data['log_file'])
    return {'msg': 'long ansible cmd has been executed'}
```

API 接口 ansible_demo/demo_2/api/api_demo_2.py:

```
@list_route(methods=['get', 'post'])
def execute_long_ansible(self, request):
    data = request.data
    f = NamedTemporaryFile(delete=False)
    data['log_file'] = f.name
    res = long_ansible_read_log.delay(data)
    return Response({'task_id': res.task_id, 'log_file': f.name})
```

步骤 3: 定义绑定 url。

在 ansible_demo/demo_2/views.py 中添加如下内容:

```
@api_view(['GET', 'POST'])
def demo_read_log(request):
    if request.method == 'GET':
        return render(request, 'demo_2/defines/demo_read_log.html')
```

步骤 4: 定义被绑定 url。

在 ansible_demo/demo_2/urls.py 中添加如下内容:

```
url(r'^demo_read_log/', demo_read_log),
```

步骤 5: 定义 HTML 基础页面。

读取日志文件的 HTML 基础定义文件 ansible_demo/root_demo/templates/ demo_2/defines/ demo_read_log.html。

```
{% extends "normal/base.html" %}

{% load staticfiles %}
{% block title %}demo_read_log{% endblock %}

{% block content %}
{% include "demo/demo_read_log.html" %}
{% endblock %}

{% block chosen_js_file %}
<script type="text/javascript" src="{% static 'js/angular-controller/demo_
ansible/demo_read_log_ctrl.js' %}"></script>
<!-- end -->

{% endblock %}
```

步骤 6: 编写 HTML 主页面。

ansible_demo/root_demo/templates/demo_2/pages/demo_read_log.html 文件如下:

```
<div class="col-md-12" ng-controller="demo_read_log_ctrl">
  <br/>
  <div class="form-group col-md-offset-4">
    <label class="col-md-2 control-label text-muted big"> 输入 yml 文件名 :</label>
    <div class="col-md-6">
      <input class="form-control" ng-model="yaml_file" ng-init="yaml_file='long_read_log.yml'">
    </div>
  </div>
  <div class="row col-md-12">
    <br />
    <button class="btn btn btn-primary col-md-offset-5" ng-click="execute_read()" ng-show="!read_flag"> 开始执行任务 </button>
    <button class="btn btn btn-primary col-md-offset-5" ng-show="read_flag" disabled> 开始执行任务 </button>
    <button class="btn btn btn-primary col-md-offset-1" ng-click="revoke_task(task_id)" ng-show="task_id" ng-if="read_flag"> 停止任务 </button>
  </div>
  <br/>
  <div class="col-md-12">
    <br/>
    <p class="col-md-offset-4" ng-if="task_id">task_id: (( task_id ))</p>
    <br/>
    <p class="col-md-offset-4" ng-if="state"> 任务状态: (( state ))</p>
    <br/>
    <label class="col-md-offset-4 control-label" ng-if="ansible_log">ansible 日志 :</label>
    <br/>
    <textarea class="form-control" rows="20" ng-model="ansible_log" ng-show="ansible_log" id="textscroll"></textarea>
  </div>
</div>
```

步骤 7: 编写注入的 JavaScript 文件。

下面来编写注入的 JavaScript 文件 ansible_demo/root_demo/static/js/angular-controller/demo_2/demo_read_log_ctrl 文件的 part1。

Part1 作用如下。

1) 用来获取 task_id 并运行后台测了 celery 任务, 使用的 API 为 /demo_2/demo2_api/execute_long_ansible/, 同时生产临时文件来记录日志。

2) 初次读取日志并为它传入初始值 (使用方法 read_log({'seek': 0, 'task_id': result['task_id'], 'log_file': result['log_file']}))。

Part1: 部分代码如下所示。

```
app.controller('demo_read_log_ctrl',function($scope, $http){
```

```

$scope.read_flag = false;
$scope.execute_read = function() {
    $scope.ansible_log = '';
    $scope.read_flag = true;
    console.log($scope.read_flag);
    $scope.state = 'PENDING';
    data = {'yaml_file': $scope.yaml_file};
    $http.post('/demo_2/demo2_api/execute_long_angular/' , data)
    .success(function(result){
        $scope.task_id = result['task_id'];
        read_log({'seek': 0 , 'task_id': result['task_id'], 'log_file': result['log_file']});
        $scope.ansible_log += '\nstart read\n';
    }).error(function(err){
        console.log(err)
    });
};
.....

```

代码中主要使用的 HTML 的绑定数据如下：

- ❑ `$scope.yaml_file` 为我们要执行的 YAML 文件；
- ❑ `$scope.read_flag` 为是否读取日志的标志，为 `False` 将停止读取日志；
- ❑ `$scope.ansible_log` 为日志的内容，将使用增量的方式添加到这个参数；
- ❑ `$scope.state` 为 `celery` 任务的标志，设定当为 `SUCCESS` 或 `FAILURE` 时停止读取；
- ❑ `$scope.task_id` 为任务的 id，通过记录这个值，来获取关于这个任务的信息。

每次读取日志时我们都会记录日志文件在服务器上的位置，并记录每次读取的位置，以便下次从该位置开始读取，以减少服务器与浏览器之间的数据传输。

`ansible_demo/root_demo/static/js/angular-controller/demo_2/demo_read_log_ctrl` 为文件的 Part2。

Part2: 我们定义了每隔 5 秒读入日志的方法，并能实时刷新日志，让日志自行往下滚动。

```

function read_log(data){
    $http.post('/demo_2/demo2_api/read_long_angular/' , data)
    .success(function(result){
        console.log(result);
        data = result;
        $scope.ansible_log += result['logs'];
        if($scope.state == 'REVOKED'){
            $scope.ansible_log += '\nend read\n';
            $scope.read_flag = false;
            return;
        }
        $scope.state = data['state'];
        $scope.read_flag = data['read_flag']
        if($scope.read_flag == false){
            alert('task ' + result['task_id'] + ' is over!');
            document.getElementById("textscroll").scrollTop=document.

```



```

getElementById("textscroll").scrollHeight;
$scope.ansible_log += '\nend read\n';
}else{
    if($scope.read_flag == false){
        alert('task ' + result['task_id'] + ' is over')
        document.getElementById("textscroll").scrollTop=document.
        getElementById("textscroll").scrollHeight;
        $scope.ansible_log += '\nend read\n';
    }else{
        setTimeout(function() { read_log(data)}, 5000); //wait every
        5 second to read log
        if(document.getElementById("textscroll").scrollTop + 1000>=
        document.getElementById("textscroll").scrollHeight ||
        document.getElementById("textscroll").scrollTop == 0){
            setTimeout(function(){
                document.getElementById("textscroll").scrollTop=document.
                getElementById("textscroll").scrollHeight;
            },100);
        }
    }
}
}).error(function(err){
    console.log('err')
});
}

```

其中，read_log 函数中的传参 data 值每次都需要 3 个值的 JSON 格式：

{'seek': xxx, 'task_id': xxx, 'log_file': xxx}。即：

- 1) 此次日志文件读取的位置：seek;
- 2) 任务的 id：task_id;
- 3) 日志文件的位置：log_file。

我们调用的 API 为 /demo_2/demo2_api/read_long_ansible/。

```

def check_task_end(self, task_id):
    res = AsyncResult(task_id)
    return res.state

@list_route(methods=['get', 'post'])
def read_long_ansible(self, request):
    data = request.data
    if not data.has_key('log_file'):
        log_file = '/tmp/ansible_long.log'
    else:
        log_file = data['log_file']
    if not os.path.exists(log_file):
        data['read_flag'] = True
        data['logs'] = ''
    return Response(data)

```

```

with open(log_file, 'r') as f:
    f.seek(data['seek'])
    data['logs'] = f.read()
    data['seek'] = f.tell()
data['state'] = self.check_task_end(data['task_id'])
if data['state'] in ['SUCCESS', 'FAILURE', 'REVOKED']:
    data['read_flag'] = False
else:
    data['read_flag'] = True
return Response(data)

```

这个 API 比较简单，我们仍然返回 `{'seek': xxx, 'task_id': xxx, 'log_file': xxx}`，后续我们会反复调用这个 API，通过“`setTimeout 5000`”，来设定每 5 秒读取一次。

```
setTimeout(function() { read_log(data)}, 5000);
```

另外，要想让日志自动向下滚动，还要在对应的 HTML 的 `textarea` 设定对应的 id 为 `textscroll`。

对应的 HTML:

```
<textarea class="form-control" rows="20" ng-model="ansible_log" ng-show="ansible_log" id="textscroll"></textarea>
```

对应的 JavaScript:

```
document.getElementById("textscroll").scrollTop=document.getElementById("textscroll").scrollHeight;
```

设置滚动条不到底部不自动向下滚动:

```

if(document.getElementById("textscroll").scrollTop + 1000>=document.
getElementById("textscroll").scrollHeight || document.
getElementById("textscroll").scrollTop == 0){
setTimeout(function(){
    document.getElementById("textscroll").scrollTop=document.
getElementById("textscroll").scrollHeight;
},100);
}

```

Part3: 停止任务的 JavaScript。

```

$scope.revoke_task = function(task_id){
    data = {'task_id': task_id}
    $http.post('/demo2/demo2_api/long_ansible_revoke/', data)
    .success(function(result){
        alert('任务已经停止')
        $scope.task_id = '';
        $scope.state = 'REVOKED';
    }).error(function(err){
        console.log('err')
    });
};

```

我们使用的回收任务的 API 为上一节的方法 `/demo_2/demo2_api/long_ansible_revoke/`，我们所需要的参数只有 `task_id`。

步骤 8：完成编写并测试。

我们就完成了修改主机的 `hosts` 文件的页面，现在来访问它。

`http://youraddress:yourport/demo_2/demo_read_log/`

最后，我们还有一个任务未完成，就是通过网页给 YML 传入参数，这个只需要结合上一章节的内容来修改即可。在稍后的章节中也会使用到，所以在该内容不会再过于复杂地嵌入其中。想要得到最后的案例。请跳到最后一节。

14.4 通过页面上上传文件并基于 Ansible 分发

在这一节里，我们主要学习如何通过页面上传文件，如图 14-7 所示。进一步调用 Ansible 命令，来分发上传的文件到各个服务器上。

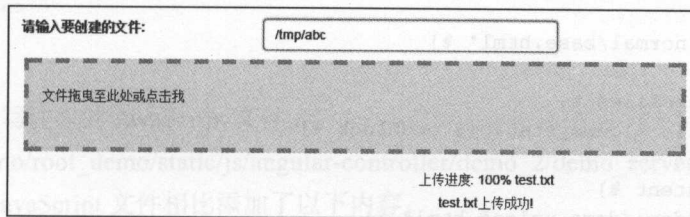


图 14-7 文件上传 Web 页面

步骤 1：该示例未用到 YML 文件，跳过。

步骤 2：使用的 API 接口。该 API 接口主要是储存上传的文件到服务器的自定义目录中去。

在 `ansible_demo/demo_2/api/api_demo2.py` 中添加：

```
@list_route(methods=['get', 'post'])
def file_upload(self, request):
    all_str = request.data['para']
    all_data = json.loads(all_str)
    saved_file_name = all_data['saved_file_name']      # 保存的文件名
    saved_file_dir = os.path.dirname(saved_file_name)
    filename = request.data['file']
    file_name = str(filename)
    if not os.path.exists(saved_file_dir):             # 目录不存在时，创建它
        os.makedirs(saved_file_dir)
    try:
        destination = open(saved_file_name, 'wb+')
        for chunk in filename.chunks():
            destination.write(chunk)
        destination.close()
    return Response({'flag': True})
```

```
except:
    return Response({'flag': False})
```

步骤 3: 定义绑定 url。

在 `ansible_demo/demo_2/views.py` 中添加:

```
@api_view(['GET', 'POST'])
def demo_upload(request):
    if request.method == 'GET':
        return render(request, 'demo_upload.html')
```

步骤 4: 定义被绑定 url。

在 `ansible_demo/demo_2/urls.py` 中添加:

```
url(r'^demo_upload/', demo_upload),
```

步骤 5: 定义 HTML 基础页面。

上传文件的基础 HTML 的结构定义文件 `ansible_demo/root_demo/templates/demo_upload.`

html 如下:

```
{% extends "normal/base.html" %}

{% load staticfiles %}
{% block title %}demo_static{% endblock %}

{% block content %}
{% include "demo/demo_upload.html" %}
{% endblock %}

{% block chosen_js_file %}

<script type="text/javascript" src="{% static 'js/angular-controller/demo_
ansible/demo_upload_ctrl.js' %}"></script>
<!-- end -->

{% endblock %}
```

步骤 6: 上传的 HTML 文件 `ansible_demo/root_demo/templates/demo/demo_upload.html` 如下:

```
<div class="col-md-12 column">
  <div class="col-md-offset-3 column">
    <div ngf-drop ngf-select ng-model="upload_file" ng-bind="a=' 文件拖曳
    至此处或点击我 '" class="drop-box2 col-md-12 column"
    ngf-drag-over-class="dragover" ngf-multiple="false" ngf-allow-dir="true"
    accept="image/*,application/pdf"></div>
    <div ngf-no-file-drop>File Drag/Drop is not supported for this browser</
    div>
    <div class="col-md-12 column">
      <div ng-if="proc_log">
        <h5 class="text-center text-success">((proc_log))</h5>
```



```

    </div>
    <div ng-show="flag">
      <h5 class="text-center text-success">{{msg}}</h5>
    </div>
    <div ng-show="!flag">
      <h5 class="text-center text-danger">{{msg}}</h5>
    </div>
  </div>
</div>

<style>
.drop-box2 {
  background: # F8F8F8;
  border: 5px dashed # DDD;
  width: 70%;
  height: 100px;
  padding-top: 25px;
  margin: 10px;
}
</style>
</div>

```

步骤 7: 编写注入的 JavaScript 文件。

ansible_demo/root_demo/static/js/angular-controller/demo_2/demo_server_modify_ctrl.js 与之前 deploy 的 JavaScript 文件相比添加了以下内容:

```

app.controller('demo_upload_ctrl', function($scope, $http, Upload){
  function goupload(url) {
    console.log($scope.upload_file)
    if(!$scope.upload_file) {
      return;
    }else{
      if(!$scope.filename){
        alert('文件未命名, 请先输入要创建的文件名');
        return;
      }else{
        Upload.upload({
          url: url,
          data: {file: $scope.upload_file, 'para': JSON.stringify($scope.para)},
        }).then(function (resp) {
          console.log(resp)
          $scope.flag = resp['data']['flag'];
          if (resp['data']['flag']) {
            $scope.msg = resp.config.data.file.name + '上传成功!';
          } else {
            $scope.msg = '上传出错: ' + resp['data']['output'];
          }
        })
      }
    }
  }
}

```

```

        $scope.filename = '';
    }, function (resp) {
        console.log('Error status: ' + resp.status);
    }, function (evt) {
        var progressPercentage = parseInt(100.0 * evt.loaded / evt.total);
        $scope.proc_log = '上传进度: ' + progressPercentage + '% ' +
            evt.config.data.file.name;
    });
}
}
});

$scope.filename = '';
$scope.$watch('upload_file', function () {
    $scope.para = {'saved_file_name': $scope.filename}
    goupload('/demo_2/demo2_api/file_upload/');
}, true);
});

```

步骤 8: 完成编写并测试。

我们就完成了文件的上传，现在来访问它。

`http://youraddress:yourport/demo_2/demo_upload/`

之后我们可以使用 Ansible 来调用分发，由于篇幅有限，就不编写了，只要在 YML 文件中使用 Ansible 的 `copy` 命令即可。

14.5 在页面上构建 YML 文件注册中心

由于应用的不断增多，需求的不断改变，我们不可能为每一个需求去增加或者修改代码，这样编出的代码将是很糟糕的。所以我们就需要一个类似于注册中心的页面，来配置我们的 YML，以便其他同事可以使用。

首先，我们要为这些注册的信息来建立一个数据库。

在 `ansible_demo/demo_2/api/models.py` 中添加：

```

class Ansible_Yml_Register(models.Model):
    yml_file = models.CharField(max_length=200, blank=True, default='')
    # 注册的 YML 文件
    yml_maintenancer = models.CharField(max_length=50, blank=True, default='')
    # 注册 YML 的维护人
    yml_parameter = models.TextField(blank=True, null=True) # 可接受的参数
    accept_host_group = models.CharField(max_length=200, blank=True, default='')
    # YML 可接受的 hosts 组
    comment = models.CharField(max_length=200, blank=True, default='')
    # YML 的使用说明
    register_time = models.DateTimeField(auto_now_add=True) # 注册时间

```

之后，我们为数据库添加序列化，便于调取数据，如果读者有不清楚该部分，可以到上一节学习该内容。

在 `ansible_demo/demo_2/api/serializers.py` 中添加以下代码：

```
class Ansible_Yml_RegisterSerializer(serializers.ModelSerializer):  
    class Meta:  
        model = Ansible_Yml_Register
```

完成了数据库部分后，就可以开始 YML 注册中心的编写工作了。先看下我们要完成的效果，注册 YML 的页面如图 14-8 所示，添加一项 YML 注册的页面如图 14-9 所示。

添加一项 yml						
id	注册 yml 文件	host 组	维护人	参数列表	说明	注册时间
1	echo_choice_x.yml	group-test1	master	[{"id":"389e8fb8-20a3-bf94-d292-9b70a40e7cc7","name":"choice","type":"choice","values":"1in2","comment":"test choice"}]	choice 远程 echo 测试	2016-06-23T14:17:49Z
2	echo_input_x.yml	group-test1	developer	[{"id":"b84bf3d0-d613-713d-911b-dac1ecd9676c","name":"input","type":"string","values":"1","comment":"test input"}, {"id":"59913a8b-0dbe-ebb2-f643-63a1f8eee5ad","name":"choice","type":"choice","values":"2","comment":"test two choice"}]	input 远程 echo 测试	2016-06-23T14:18:45Z

图 14-8 注册 YML 的页面

添加 yml 文件

注册 yml 文件:

host 组:

维护人:

添加一项参数

参数名称:

参数类型: choice

参数选项:

参数说明:

yml 使用说明:

确认添加

取消

图 14-9 添加一项 YML 注册的页面

简单地介绍下我们页面的主要功能。

- 1) 记录 YML 文件的地址。
- 2) 设定该 YML 可以使用的 host 组（这时候我们 14.1 节完成的功能就将派上用处了）。
- 3) 添加所有的 YML 中用到的参数，目前设定有 string 字符型，自己填写的空格，以及 choice 选择型，并填写默认值（选项以空行隔开）。最后为该参数写一下说明即可。
- 4) 标注该脚本的作用。

先来看下我们设定的 YAML 文件的基本格式。

以下一部分原则上不做更改，只要是为了传入 hosts。

```
- hosts: "{{ansible_hosts}}"
  gather_facts: no
```

之后的 YAML 的部分，当每遇到一个 `{{}}` 变量的时候，就应该到我们的注册页面的参数添加项去注册该参数，为该参数指定数据类型（string 或者 choice）和默认值。

步骤 1：该示例未用到 YAML 文件，跳过。

步骤 2：使用的 API 接口。

```
@list_route(methods=['get', 'post'])
def create_yaml_file_define(self, request):
    data = request.data
    Ansible_Yml_Register.objects.create(**data)
    return Response({'flag': True})

@list_route(methods=['get', 'post'])
def delete_yaml_file_define(self, request):
    data = request.data
    Ansible_Yml_Register.objects.filter(id=data['remove_id']).delete()
    return Response({'flag': True})
```

```
@list_route(methods=['get', 'post'])
def update_yaml_file_define(self, request):
    data = request.data
    Ansible_Yml_Register.objects.filter(id=data['id']).update(**data)
    return Response({'flag': True})
```

步骤 3：定义绑定 url。

在 `ansible_demo/demo_2/views.py` 中添加：

```
@api_view(['GET', 'POST'])
def demo_config_center(request):
    if request.method == 'GET':
        return render(request, 'demo_2/defines/demo_config_center.html')
```

步骤 4：定义被绑定 url。

在 `ansible_demo/demo_2/urls.py` 中添加以下代码：

```
url(r'^demo_config_center/', demo_config_center),
```

步骤 5：定义 HTML 基础页面。

`ansible_demo/root_demo/templates/demo_config_center.html` 文件如下：

```
{% extends "demo_2/base/base.html" %}

{% load staticfiles %}
```



```
{% block title %}demo_config_center{% endblock %}

{% block content %}
{% include "demo_2/pages/demo_config_center.html" %}
{% endblock %}

{% block chosen_js_file %}
<script type="text/javascript" src="{% static 'js/angular-controller/demo_2/
demo_config_center_ctrl.js' %}"></script>
<!-- end -->

{% endblock %}
```

步骤 6: 编写 HTML 主页面。

ansible_demo/root_demo/templates/demo_2/pages/demo_config_center.html 文件如下。

Part1: 罗列所有可执行的 YML 文件, 根据参数类型来形成填写选项。

```
<div class="col-md-12" ng-controller="demo_operate_interface_ctrl">
  <br/>
  <div ng-repeat="i in data" class="well col-md-12">
    <div class="form-group col-md-12" >
      <label class="col-md-4 control-label text-muted">yml 文件名: ((i.yml_file))</label>
      <label class="col-md-2 control-label text-muted">维护人: ((i.yml_maintenancer))</label>
      <label class="col-md-6 control-label text-muted">使用说明: ((i.comment))</label>
    </div>
    <div class="col-md-12" >
      <div ng-repeat="j in i.yml_parameter" >
        <div class="form-group col-md-4" ng-if="j.type == 'choice'">
          <label class="col-md-4 control-label text-muted">((j.name))</label>
          <div class="col-md-8">
            <select ng-options="o for o in j.values" class="form-control"
              ng-model="i['operate'][(j.name)]" data-toggle="tooltip" data-placement="top" title="((j.comment))">
              <option disabled>请选择</option>
            </select>
          </div>
        </div>
        <div class="form-group col-md-4" ng-if="j.type == 'string'">
          <label class="col-md-4 control-label text-muted">((j.name))</label>
          <div class="col-md-8">
            <input class="form-control" ng-model="i['operate'][(j.name)]"
              data-toggle="tooltip" data-placement="top" title="((j.comment))">
          </div>
        </div>
      </div>
    </div>
  </div>
```

```
</div>
```

Part2: 任务开始按钮、结束按钮和任务执行的状态。

```
<div class="row col-md-12">
  <button class="btn btn btn-primary col-md-offset-5" data-toggle="modal"
data-target="# check_servers_before_operate" ng-click="fill_data_func(i)" ng-
show="!read_flag"> 开始执行任务 </button>
  <button class="btn btn btn-primary col-md-offset-5" ng-show="read_flag"
disabled> 开始执行任务 </button>

</div>
<br/>
</div>
<button class="btn btn btn-primary col-md-offset-5" ng-click="revoke_
task(task_id)" ng-show="task_id" ng-if="read_flag"> 停止任务 </button>

<br/>
<div class="col-md-12 well" ng-if="ansible_log">
  <br/>
  <p class="col-md-offset-4" ng-if="yaml_file">执行的yaml文件: (( yaml_file ))</p>
  <br/>
  <p class="col-md-offset-4" ng-if="state">任务状态: (( state ))</p>
  <br/>
  <label class="col-md-offset-4 control-label" ng-if="ansible_log">ansible
日志:</label>
  <br/>
  <textarea class="form-control" rows="20" ng-model="ansible_log" ng-
show="ansible_log" id="textscroll"></textarea>
</div>
```

Part3: 选择可执行的主机，把之前存入数据库的主机组显示出来，供用户进行选择。

```
<br/>
<!-- Modal -->
<div class="modal fade" id="check_servers_before_operate" tabindex="-1"
role="dialog" aria-labelledby="myModalLabel" aria-hidden="true">
  <div class="modal-dialog modal-lg" style="width: 1200px; overflow-y: scroll;
max-height:85%; margin-top: 5px; margin-bottom:5px;">
    <div class="modal-content">
      <div class="modal-header" class="col-md-12">
        <button type="button" class="close" data-dismiss="modal" aria-
label="Close"><span aria-hidden="true">&times;</span></button>
        <div class="col-md-offset-6">
          <b class="modal-title text-center">请选择机器，并将执行发布 </b>
        </div>
      </div>
      <div class="container">
        <div class="modal-body" class="col-md-12">
          <div class="col-md-12">
            <span ng-repeat="j in select_servers" class="col-md-4">
```

```

        <input type="checkbox" class="css-checkbox"
        id='select_((j.name))' name="select_((j.name))"
        ng-model="j.checked" ng-checked="j.checked">
        <label for="select_((j.name))" class="css-
        label"><small>((j.name))</small></label>
    </span>
    <br/>
</div>
</div>
<br/>
<br/>
<div class="modal-footer col-md-12">
    <br/>
    <div class="col-md-12">
        <div class="col-md-12 text-center">
            <button type="button" class="btn btn-primary" data-
            dismiss="modal" ng-click="execute_read()"> 确认发布 </button>
            <button type="button" class="btn btn-default" data-
            dismiss="modal"> 取消 </button>
        </div>
    </div>
</div>
</div>
</div>
</div><!-- /.modal-content -->
</div><!-- /.modal-dialog -->
</div><!-- /.modal -->

</div>

```

步骤 7: 编写注入的 JavaScript 文件。

下面来编写用于注入的 JavaScript 文件, 本例中为 `ansible_demo/root_demo/static/js/angular-controller/demo_2/demo_config_center_ctrl.js` 文件。我们来逐一剖析各个部分。

Part1: 修改信息后及时更新 HTML 页面的数据函数 `get_all_data`。

```

app.controller('demo_config_center_ctrl', function($scope, $http) {
    get_all_data = function () {
        $http.get('/demo_2/ansible_yaml_register_api/')
        .success(function (res) {
            console.log(res);
            _.each(res, function(i){
                i['yaml_parameter'] = JSON.parse(i['yaml_parameter']);
            })

            $scope.data = res;
        })
    }
    get_all_data()

```

Part2: 为弹出框创建一个 `tag(create, modify, remove)` 来生成对应的弹出框的 JavaScript 功能。

```

$scope.create_tag = function(tag, ins){
    $scope.operate_type = tag;
    if(tag == 'create'){
        $scope.yml_parameter = [];
    }
    if(tag == 'remove' || tag == 'modify'){
        $scope.id = ins.id;
        $scope.yml_file = ins.yml_file;
        $scope.yml_maintenancer = ins.yml_maintenancer;
        $scope.yml_parameter = ins.yml_parameter;
        $scope.accept_host_group = ins.accept_host_group;
        $scope.comment = ins.comment;
    }
};

```

Part3：产生 guid 的函数。由于我们把所有的参数选项框的数据用 JSON 格式存入一个字段中，通过 guid 来标记这个参数选项框，通过这个 id 来查找这个参数选项框，来执行更新和删除操作。

```

function guid() {
    function s4() {
        return Math.floor((1 + Math.random()) * 0x10000)
            .toString(16)
            .substring(1);
    }
    return s4() + s4() + '-' + s4() + '-' + s4() + '-' +
        s4() + '-' + s4() + s4() + s4();
}

```

Part4：添加一项参数选项框的函数。

```

$scope.add_more_parameter_func = function(){
    $scope.yml_parameter.push({'id': guid(), 'name': '', 'type': 'choice',
        'values': '', 'comment': ''});
};

```

Part5：删除一个参数选项框的函数。

```

$scope.delete_parameter_func = function(id){
    console.log(id)
    console.log($scope.yml_parameter)
    $scope.yml_parameter = _.reject($scope.yml_parameter, function (i){ return
        i['id'] == id});
};

```

Part6：更新、删除或创建注册 YAML 的数据。

```

$scope.generate_config_func = function() {
    $scope.save_wait = true;
    if($scope.operate_type == 'create') {
        data = {'yml_file': $scope.yml_file,

```



```

        'yaml_maintenancer': $scope.yaml_maintenancer,
        'yaml_parameter': JSON.stringify($scope.yaml_parameter),
        'accept_host_group': $scope.accept_host_group,
        'comment': $scope.comment
    }
    $http.post('/demo_2/demo2_api/create_yaml_file_define/', data)
    .success(function (res) {
        if (res['flag'] == true) {
            alert('定义创建成功')
            get_all_data()
        } else {
            alert('定义创建失败')
        }
        $scope.save_wait = '';
    }).error(function (res) {
        alert('定义创建失败')
        $scope.save_wait = '';
    })
}
if($scope.operate_type == 'remove') {
    data = {'remove_id': $scope.id};
    $http.post('/demo_2/demo2_api/delete_yaml_file_define/', data)
    .success(function (res) {
        if (res['flag'] == true) {
            alert('定义删除成功')
            get_all_data()
        } else {
            alert('定义删除失败')
        }
        $scope.save_wait = '';
    }).error(function (res) {
        alert('定义删除失败')
        $scope.save_wait = '';
    })
}
if($scope.operate_type == 'modify') {
    data = {'id': $scope.id,
        'yaml_file': $scope.yaml_file,
        'yaml_maintenancer': $scope.yaml_maintenancer,
        'yaml_parameter': JSON.stringify($scope.yaml_parameter),
        'accept_host_group': $scope.accept_host_group,
        'comment': $scope.comment
    }
    $http.post('/demo_2/demo2_api/update_yaml_file_define/', data)
    .success(function (res) {
        if (res['flag'] == true) {
            alert('定义更新成功')
            get_all_data()
        } else {
            alert('定义更新失败')
        }
    }
}

```

```

        $scope.save_wait = '';
    }).error(function (res) {
        alert('定义更新失败');
        $scope.save_wait = '';
    })
}
});
```

步骤 8: 完成编写并测试。
我们就完成了修改主机的 hosts 文件的页面，现在来访问它。

http://youraddress:yourport/demo_2/demo_config_center/

14.6 操作者注册中心界面

在上一节中，我们已经完成了 YML 注册中心的配置，接下来就需要把这个配置中心的内容转化为用户或者同事能够看得懂的页面，以方便大家一起使用，用户操作界面如图 14-10 所示，操作者界面选择机器如图 14-11 所示，开始执行任务如图 14-12 所示。

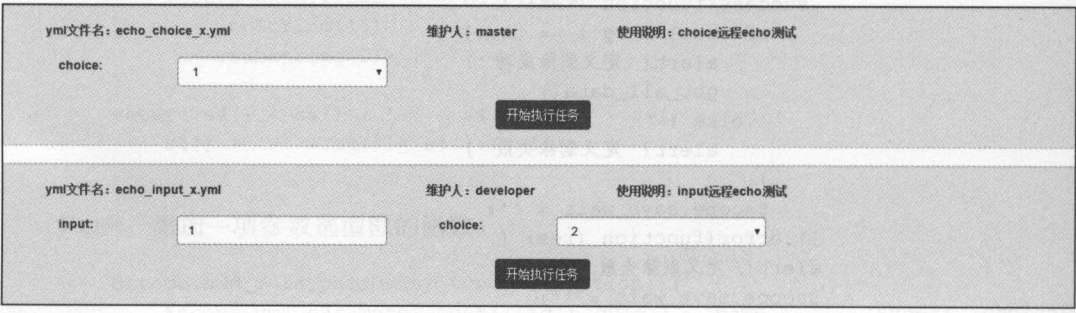


图 14-10 用户操作界面

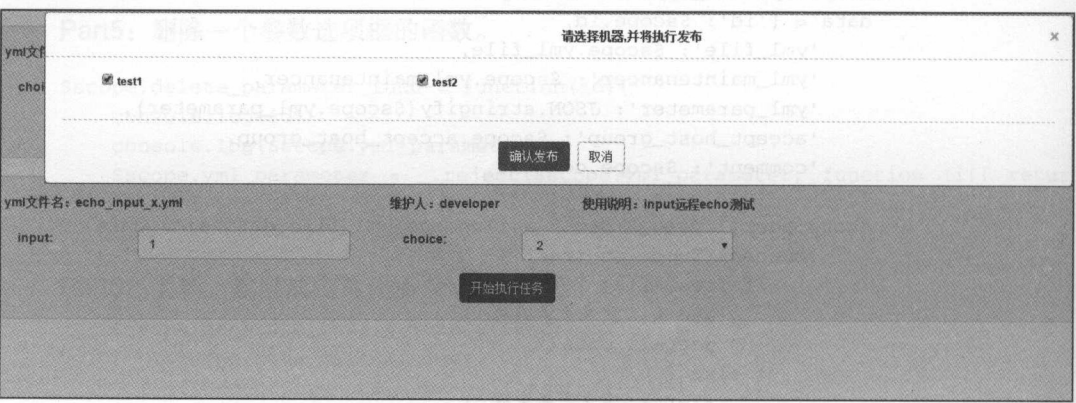


图 14-11 操作者界面选择机器

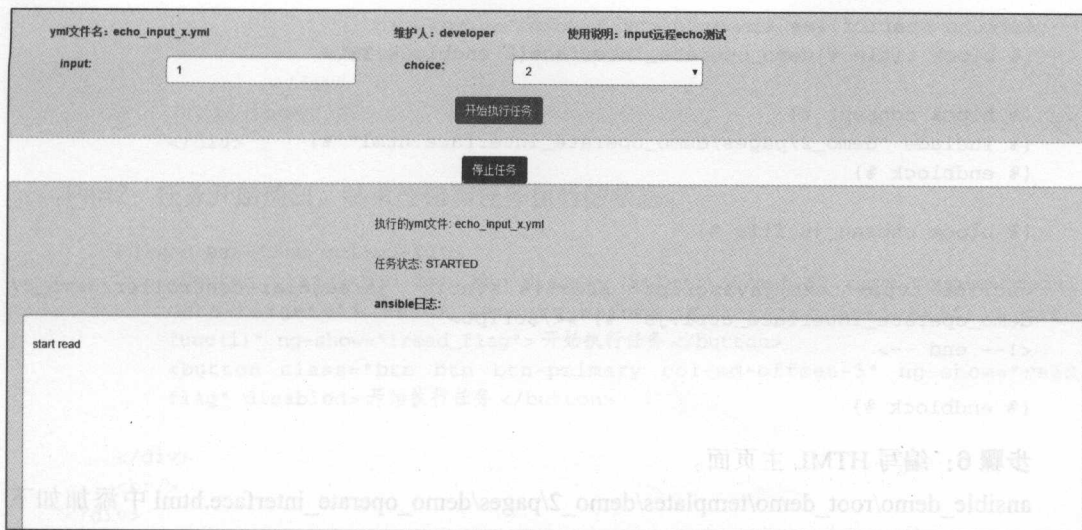


图 14-12 开始执行任务

步骤 1: 该示例未用到 YML 文件，跳过。

步骤 2: 使用的 API 接口。

```
@list_route(methods=['get', 'post'])
def execute_yml_ansible(self, request):
    data = request.data
    f = NamedTemporaryFile(delete=False)
    data['log_file'] = f.name
    res = common_ansible_bg.delay(data)
    return Response({'task_id': res.task_id, 'log_file': f.name})
```

步骤 3: 定义绑定 url。

在 `ansible_demo/demo_2/views.py` 中添加以下代码：

```
@api_view(['GET', 'POST'])
def demo_operate_interface(request):
    if request.method == 'GET':
        return render(request, 'demo_2/defines/demo_operate_interface.html')
```

步骤 4: 定义被绑定 url。

在 `ansible_demo/demo_2/urls.py` 中添加以下代码：

```
url(r'^demo_operate_interface/', demo_operate_interface),
```

步骤 5: 定义 HTML 基础页面。

在 `ansible_demo/root_demo/templates/demo_2/defines/demo_operate_interface.html` 中添加：

```
{% extends "demo_2/base/base.html" %}
```

```
{% load staticfiles %}
{% block title %}demo_operate_interface{% endblock %}

{% block content %}
{% include "demo_2/pages/demo_operate_interface.html" %}
{% endblock %}

{% block chosen_js_file %}

<script type="text/javascript" src="{% static 'js/angular-controller/demo_2/
demo_operate_interface_ctrl.js' %}"></script>
<!-- end -->

{% endblock %}
```

步骤 6: 编写 HTML 主页面。

在 `ansible_demo/root_demo/templates/demo_2/pages/demo_operate_interface.html` 中添加如下内容

Part1: 罗列所有可执行的 YAML 文件，根据参数类型来形成填写选项。

```
<div class="col-md-12" ng-controller="demo_operate_interface_ctrl">
  <br/>
  <div ng-repeat="i in data" class="well col-md-12">
    <div class="form-group col-md-12" >
      <label class="col-md-4 control-label text-muted">yml 文件名: ((i.yml_
file))</label>
      <label class="col-md-2 control-label text-muted">维护人: ((i.yml_
maintenancer))</label>
      <label class="col-md-6 control-label text-muted">使用说明: ((i.comment))</
label>
    </div>
    <div class="col-md-12" >
      <div ng-repeat="j in i.yml_parameter" >
        <div class="form-group col-md-4" ng-if="j.type == 'choice'">
          <label class="col-md-4 control-label text-muted">((j.name)):</
label>
          <div class="col-md-8">
            <select ng-options="o for o in j.values" class="form-
control" ng-model="i['operate'][j['name']] " data-toggle="tooltip"
data-placement="top" title="((j.comment))">
              <option disabled> 请选择 </option>
            </select>
          </div>
        </div>
      </div>
      <div class="form-group col-md-4" ng-if="j.type == 'string'">
        <label class="col-md-4 control-label text-muted">((j.name)):</
label>
        <div class="col-md-8">
          <input class="form-control" ng-model="i['operate'][j['name']] "
```



```

        data-toggle="tooltip" data-placement="top" title="{{ (j.comment) }}">
    </div>
</div>
</div>

```

Part2: 任务开始按钮、结束按钮和任务执行的状态。

```

<div class="row col-md-12">
    <button class="btn btn btn-primary col-md-offset-5" data-toggle="modal"
        data-target="# check_servers_before_operate" ng-click="fill_data_
        func(i)" ng-show="!read_flag">开始执行任务 </button>
    <button class="btn btn btn-primary col-md-offset-5" ng-show="read_
        flag" disabled>开始执行任务 </button>

</div>
<br/>
</div>
<button class="btn btn btn-primary col-md-offset-5" ng-click="revoke_task(task_
id)" ng-show="task_id" ng-if="read_flag">停止任务 </button>
<br/>
<div class="col-md-12 well" ng-if="ansible_log">
    <br/>
    <p class="col-md-offset-4" ng-if="yaml_file">执行的yaml文件: {{ yaml_file }}</p>
    <br/>
    <p class="col-md-offset-4" ng-if="state">任务状态: {{ state }}</p>
    <br/>
    <label class="col-md-offset-4 control-label" ng-if="ansible_log">ansible
    日志:</label>
    <br/>
    <textarea class="form-control" rows="20" ng-model="ansible_log" ng-
    show="ansible_log" id="textscroll"></textarea>
</div>

```

Part3: 选择可执行的主机，把之前存入数据库的主机组显示出来，供用户进行选择。

```

<br/>
<!-- Modal -->
<div class="modal fade" id="check_servers_before_operate" tabindex="-1" role="dialog"
aria-labelledby="myModalLabel" aria-hidden="true">
    <div class="modal-dialog modal-lg" style="width: 1200px; overflow-y: scroll;
    max-height: 85%; margin-top: 5px; margin-bottom: 5px;">
        <div class="modal-content">
            <div class="modal-header" class="col-md-12">
                <button type="button" class="close" data-dismiss="modal"
                aria-label="Close"><span aria-hidden="true">&times;</span></button>
                <div class="col-md-offset-6">
                    <b class="modal-title text-center">请选择机器，并将执行发布 </b>
                </div>
            </div>
        </div>
    </div>

```

```

<div class="container">
  <div class="modal-body" class="col-md-12">
    <div class="col-md-12">
      <span ng-repeat="j in select_servers" class="col-md-4">
        <input type="checkbox" class="css-checkbox" id='select_
          ((j.name))' name="select_((j.name))" ng-model="j.
            checked" ng-checked="j.checked">
        <label for="select_((j.name))" class="css-
          label"><small>((j.name))</small></label>
      </span>
    <br/>
  </div>
  <br/>
  <br/>
  <div class="modal-footer col-md-12">
    <br/>
    <div class="col-md-12">
      <div class="col-md-12 text-center">
        <button type="button" class="btn btn-primary" data-
          dismiss="modal" ng-click="execute_read()"> 确认发布 </button>
        <button type="button" class="btn btn-default" data-
          dismiss="modal"> 取消 </button>
      </div>
    </div>
  </div><!-- /.modal-content -->
</div><!-- /.modal-dialog -->
</div><!-- /.modal -->
</div>

```

步骤 7: 编写注入的 JavaScript 文件。

ansible_demo/root_demo/static/js/angular-controller/demo_2/operate_interface_ctrl.js 中添加以下内容。

Part1: 为 HTML 提供注册的 YAML 所有信息来生成页面。

```

app.controller('demo_operate_interface_ctrl',function($scope, $http){

  get_all_data = function () {
    $http.get('/demo_2/ansible_yml_register_api/')
    .success(function (res) {
      console.log(res);
      _.each(res, function(i){
        i['yml_parameter'] = JSON.parse(i['yml_parameter']);
        i['operate'] = {};
        _.each(i['yml_parameter'], function(j){

```

```

        if(!_isEmpty(j['values']))){
            i['operate'][j['name']] = j['values'].split('\n')[0];
            j['values'] = j['values'].split('\n')[1];
        }else{
            i['operate'][j['name']] = '';
            j['values'] = '';
        }
    }
    console.log(i['yaml_parameter'])
})
$scope.data = res;
})
}
get_all_data()

```

Part2: 通过下面的函数把已选的主机放入执行列表中。

```

$scope.fill_data_func = function(data){
    $scope.fill_data = data;
    console.log(data['accept_host_group'])
    $http.get('/demo_2/ansible_host_api/')
    .success(function (res) {
        $scope.select_servers = [];
        console.log(res)
        _.each(res, function(i){
            if(i['group'] == data['accept_host_group']){
                i['checked'] = true;
                $scope.select_servers.push(i)
            }
        })
    })
}

```

Part3: 开始执行任务，并实时读取 Ansible 的日志文件。

```

$scope.read_flag = false;
$scope.execute_read = function() {
    fill_data = angular.copy($scope.fill_data);
    fill_data['ansible_hosts'] = [];
    console.log($scope.select_servers)
    _.each($scope.select_servers, function(j){
        if(j['checked']){
            fill_data['ansible_hosts'].push(j['name']);
        }
    });
    console.log(fill_data)
    if(!_isEmpty(fill_data)){
        alert('hosts 不能为空')
        return;
    }else{

```

```

        fill_data['operate']['ansible_hosts'] = fill_data['ansible_hosts'].join(':')
    }
    $scope.ansible_log = '';
    $scope.read_flag = true;
    $scope.state = 'PENDING';
    $scope.yml_file = fill_data.yml_file;
    data = {'yml_file': fill_data.yml_file, 'operate': fill_data.operate};
    $http.post('/demo_2/demo2_api/execute.yml_ansible/', data)
    .success(function(result){
        $scope.task_id = result['task_id'];
        read_log({'seek': 0, 'task_id': result['task_id'], 'log_file': result['log_file']});
        $scope.ansible_log += '\nstart read\n';
    }).error(function(err){
        console.log(err)
    });
};

```

Part4: 调用 Ansible 日志文件读取的函数。

```

function read_log(data){
    $http.post('/demo_2/demo2_api/read_log_ansible/', data)
    .success(function(result){
        console.log(result);
        data = result;
        $scope.ansible_log += result['logs']; //add logs to model(ansible_log)
        if($scope.state == 'REVOKED'){
            $scope.ansible_log += '\nend read\n';
            $scope.read_flag = false;
            return;
        }
        $scope.state = data['state'];
        $scope.read_flag = data['read_flag']
        if($scope.read_flag == false){
            document.getElementById("textscroll").scrollTop=document.
            getElementById("textscroll").scrollHeight;
            $scope.ansible_log += '\nend read\n';
            alert('task ' + result['task_id'] + ' is over')
        }else{
            if($scope.read_flag == false){
                alert('task ' + result['task_id'] + ' is over')
                document.getElementById("textscroll").scrollTop=document.
                getElementById("textscroll").scrollHeight;
                $scope.ansible_log += '\nend read\n';
            }else{
                setTimeout(function() { read_log(data)}, 5000); //wait every 5
                second to read log
                if(document.getElementById("textscroll").scrollTop + 1000>=document.
                getElementById("textscroll").scrollHeight || document.
                getElementById("textscroll").scrollTop == 0){
                    setTimeout(function(){

```



```

        document.getElementById("textscroll").scrollTop=document.
        getElementById("textscroll").scrollHeight;
    },100);
    }
}
}).error(function(err){
    console.log('err')
});
}

```

Part5: 取消任务的函数。

```

$scope.revoke_task = function(task_id){
    data = {'task_id': task_id}
    $http.post('/demo_2/demo2_api/long_ansible_revoke/', data)
    .success(function(result){
        alert('任务已经停止')
        $scope.task_id = '';
        $scope.state = 'REVOKED';
    }).error(function(err){
        console.log('err')
    });
};

});

```

步骤 8: 完成编写并测试。

这样我们就完成了展示 Ansible 的 hosts 文件的页面，现在来访问它。

http://youraddress:yourport/demo_2/demo_operate_interface/

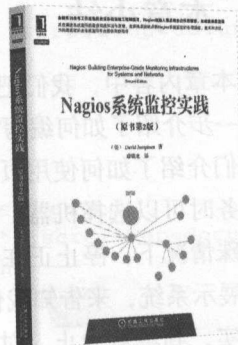
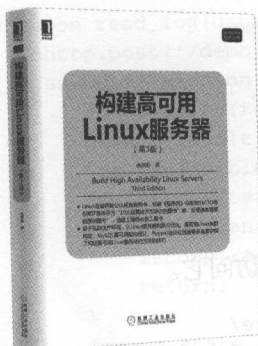
14.7 本章小结

在本章内容中，我们把所有的知识点都结合到了一起，并通过日常最可能用到的一些例子，进一步介绍了如何编写自己的自动化平台。

我们介绍了如何使用页面的方式来管理 Ansible 的 Hosts，其作用是使我们在后续执行相应的任务时可以选择机器；还介绍了 celery 这个任务管理的软件，通过它，我们可以在紧急或者特殊情况下，停止正在进行的任务，来避免我们的操作损失；另外，还配备了完整的日志实时展示系统，来告知我们任务的进展情况。这几点都是在开发平台中必须考虑的因素。

其实，到本章为止，对于 Ansible 的开发工作并未完全结束，如果想要开发可用度更高的平台的话，读者们还需开发后再配上自己的权限系统，进一步细化各人的工作职责。还要实时收集机器的所有信息，存入数据库，并到页面进行展示和告警。笔者仅仅为大家开辟一个思路，更好的思路需要通过实践和借鉴来发掘并完成，也需要大家的共同努力。

推荐阅读



作者简介

李松涛 英文昵称Stanley，2008年正式接触Linux开源领域，先后供职于上海九城、上海腾讯、上海诺亚财富等互联网公司，数次从0到1打造运维自动化体系。热衷开源技术，曾主导Ansible中文权威指南站点建设与Ansible官网本土化（<http://www.ansible.com.cn/>）工作。“运维部落”公众号发起人（迄今125+技术文章，2500人+关注），Ansible中文权威等系列开源技术QQ群发起人。



魏巍 狂热的骑行与开源爱好者，现任国内某一线互联网公司高级运维工程师。2009年开始接触并从事运维行业。专注于运维自动化、Docker及大数据领域，活跃于各大开源社区，多次应邀直播分享Linux开源技术。



甘捷 现任国内某一线互联网公司运维开发，从业以来一直专注于运维自动化开发领域，致力于提供企业级运维自动化解决方案，曾多次一力主导Web运维自动化架构设计及核心代码研发工作，结合CMDB等平台，并以Ansible作为基础支撑，不断地改善和实现运维的高自由度化、可配置化及可视化的目标。



作者结合实战经验汇总成本书，以帮助更多热爱开源的朋友。而Ansible也将成为专业人员必备技能，这本集合基础原理和实战案例的书籍会成为运维人员必备宝典。

—— **马永亮** 马哥教育创始人

在我们的客户自动化方案中，考虑到安全性、稳定性、便捷性等多方面要求，我们也把对Ansible的兼容作为首选。非常感谢Stanley和其他笔者不辞辛劳地编写此书，值得大家钦佩。相信本书能给读者带来很大的收益。

—— **王津银**（互联网运维杂谈老王）优维科技创始人

Ansible入门容易精通难，很高兴看到李松涛和他的朋友们撰写的这本书的出版，本书使快速精通Ansible成为可能。相信通过阅读本书，没有接触过Ansible的读者可以快速入门，已经在使用Ansible的读者可以从中学到更多知识。

—— **肖力**《深度实践KVM》作者

本书对Ansible的周边扩展介绍得比较实在，理论联系实践。作者从丰富的工作经验总结出案例，详细列举了celery、模块扩展等具体应用，让Ansible更加贴合实际的应用场景。

—— **张志浩** 腾讯游戏运营规划专家

“授人以鱼，不如授人以渔”，本书不但介绍了Ansible的基础知识，还介绍了Ansible的实践经验和高阶的二次开发，对读者深入理解Ansible、构建自动化运维体系非常有帮助。

—— **智锦** 资深运维从业者，杭州云霁科技有限公司CEO

